Securing untrusted memories in embedded systems

Vig, Saru

2020

Vig, S. (2020). Securing untrusted memories in embedded systems. Doctoral thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/138253>

<https://doi.org/10.32657/10356/138253>

This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License (CC BY-NC 4.0).

Downloaded on 26 Apr 2023 22:53:25 SGT

Securing Untrusted Memories in Embedded Systems

**NANYANG
TECHNOLOGICAL
UNIVERSITY**

SINGAPORE

Saru Vig

School of Computer Science and Engineering

A thesis submitted to the Nanyang Technological University
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

2020

Statement of Originality

I hereby certify that the work embodied in this thesis is the result of original research, is free of plagiarised materials, and has not been submitted for a higher degree to any other University or Institution.

30 March 2020

.....

Date

.....

Saru Vig

Supervisor Declaration Statement

I have reviewed the content and presentation style of this thesis and declare it is free of plagiarism and of sufficient grammatical clarity to be examined. To the best of my knowledge, the research and writing are those of the candidate except as acknowledged in the Author Attribution Statement. I confirm that the investigations were conducted in accord with the ethics policies and integrity standards of Nanyang Technological University and that the research data are presented honestly and without prejudice.

30 March 2020

.....

Date

.....

Dr. Lam Siew-Kei

Authorship Attribution Statement

This thesis contains material from four papers published in the following peer-reviewed journals / conferences in which I am listed as an author.

Chapter 3 is published in [1] as [Vig, Saru, Tan Yng Tzer, Guiyuan Jiang, and Siew-Kei Lam. "Customizing skewed trees for fast memory integrity verification in embedded systems." 2017 IEEE Computer Society Annual Symposium on VLSI \(ISVLSI\), pp. 213-218. IEEE, 2017.](#)

The explicit contributions of the co-authors are listed below:

- The initial project direction was provided by Dr. Siew-Kei Lam and he further supervised in editing the manuscript drafts.
- I designed the technique and prepared the manuscript draft.
- Tan Yng Tzer assisted in the experiments.
- Guiyuan Jiang helped in developing the technique for skewing.

Chapter 4 is published in [2] as [Vig, Saru, Guiyuan Jiang, and Siew-Kei Lam. "Dynamic skewed tree for fast memory integrity verification." 2018 Design, Automation and Test in Europe Conference & Exhibition \(DATE\). IEEE, 2018.](#)

The contributions of the co-authors are as follows:

- I developed and evaluated the technique, and prepared the manuscript draft.
- Guiyuan Jiang participated in the idea generation and brain-storming sessions.
- Dr. Siew-Kei Lam reviewed the manuscript and the research to identify shortcomings.

Chapter 5 is published in [3] as Vig, Saru, Rohan Juneja, Guiyuan Jiang, Siew-Kei Lam, and Changhai Ou. "Framework for fast memory authentication using dynamically skewed integrity tree." IEEE Transactions on Very Large Scale Integration (VLSI) Systems 27, no. 10 (2019): 2331-2343.

The contributions of the co-authors are as follows:

- I developed and evaluated the technique, and prepared the manuscript draft.
- Rohan Juneja assisted in developing the simulation and conducting the experiments.
- Guiyuan Jiang and Changhai Ou participated in the discussions and brain-storming sessions to identify any shortcoming.
- Dr. Siew-Kei Lam reviewed the manuscript and the research to identify shortcomings and further scope of improvements.

Part of Chapter 6 is accepted in [4] as Vig, Saru, Rohan Juneja, and Siew-Kei Lam. "DISSECT: Dynamic Skew-and-Split Tree for Memory Authentication." 2020 Design, Automation and Test in Europe Conference Exhibition (DATE). IEEE, 2020

The contributions of the co-authors are as follows:

- I developed and evaluated the technique, and prepared the manuscript draft.
- Rohan Juneja assisted in developing the simulation and conducting the experiments.
- Dr. Siew-Kei Lam reviewed the manuscript and the research to identify shortcomings and further scope of improvements.

A part of our work is published in [5] as Vig, Saru, Sarani Bhattacharya, Debdeep Mukhopadhyay, and Siew-Kei Lam. "Rapid detection of Rowhammer attacks using dynamic skewed hash tree." Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy, p. 7. ACM, 2018. However, this work is not discussed in this thesis in order to streamline and maintain focus of the thesis.

The contributions of the co-authors are as follows:

- I developed and evaluated the technique, and prepared the manuscript draft.

- Sarani Bhattacharya participated in the idea generation and brain-storming sessions.
- Dr. Siew-Kei Lam and Prof. Debdeep Mukhopadhyay reviewed the manuscript and the research to identify shortcomings.

30 March 2020

.....

Date

.....

Saru Vig

Acknowledgments

Firstly, I would like to express my deepest gratitude to my advisor Dr. Lam Siew-Kei for his patience and continuous support during my doctoral thesis. All through my PhD, he has guided me at each step from delving deeper into technical issues, providing a high-level view of my research, to editing my manuscripts. My thesis would not have been the same without his supervision.

Besides my advisor, I would like to sincerely thank Rohan Juneja, who has helped me to construct the way of my work and research efficiently. I would also like to express gratitude to Dr. Guiyuan Jiang for his continuous advice.

Next, I would like to thank my friends at NTU - Nimisha, Kratika, and Tushar, who have always provided me guidance and helped me through all hurdles.

Finally, I would like to thank my family: my grandparents, parents, and sister. They have always encouraged me and stood by me in whatever I pursue. Their affection and support for me, during my PhD and even otherwise in my life, is immeasurable. Special thanks to Vardhman, and Prerna, who have been like family to me these past four years in Singapore and have made this journey immensely memorable.

This accomplishment would not have been possible without all these people. Thank you.

Abstract

Embedded systems have become a pervasive part of our lives and they are now the driving force behind technological advancements in many commercial sectors such as health, automotive, manufacturing, etc. Due to our high dependability on such systems, it is essential that they are secure. However, commercially available embedded processors cannot guarantee secure storage in external memories, thereby exposing the vulnerability of the system to memory attacks. Protecting embedded systems from such malicious attacks poses an enormous challenge as existing memory security schemes contribute to excessive runtime overheads. This thesis aims to address this challenge by proposing compile-time and runtime strategies for securing untrusted memories from active memory attacks (e.g. replay, spoofing and splicing attacks) that can compromise the confidentiality and integrity of the memory data. Architecture-aware algorithms are proposed to support the adaptation of memory integrity trees to application workloads. Security analysis is undertaken to ensure that the performance gains are achieved without compromising on the security strength.

In order to investigate the potential of customizing memory integrity trees to application workloads for reducing the authentication overhead, we first devised an approach to restructure the integrity trees at compile-time by analyzing the static memory access patterns. The nodes of the memory integrity tree are placed on different levels of the tree based on the frequency of memory accesses. We validated the effectiveness of the proposed approach on the Altera NIOS II processor with an external DRAM. Experimental results using applications from widely-used benchmarks show that the proposed approach leads to an average runtime improvement of 18% over conventional memory protection schemes.

Since not all application workloads are known at compile-time, we proposed a technique to dynamically customize the integrity tree based on the runtime memory access

patterns. The proposed scheme enables the memory integrity tree to be dynamically restructured to significantly reduce the number of verification steps for the frequently accessed memory blocks, leading to overall reduction in the overhead for memory authentication. Experimental results show an average performance improvement of 30% compared to the conventional balanced integrity tree approach.

Next, we proposed a memory authentication framework that combines architecture-specific optimizations of the integrity tree with mechanisms that enable it to restructure at runtime based on memory access patterns. The integrity tree structure is customized based on the cache configuration in order to minimize the performance and energy overhead through speculative authentication. The framework is simulated with Multi2Sim targeting an Intel x86 processor to explore the impact of various cache configurations on the memory authentication overhead. Experimental results show that an average performance gain of 30% can be obtained over the conventional balanced tree approach. In addition, the proposed framework enables the selection of optimal cache configurations that can lead to reduced energy savings of over 40% without sacrificing performance.

Finally, we proposed three schemes to improve the scalability of our dynamic skewed integrity tree approach. The first scheme utilizes a dedicated Tree-Cache to cache memory integrity tree nodes. The tree is initialized using Van-Emde Boas (vEB) organization to take advantage of locality of reference. This led to a performance gain of approximately 15%. In the second scheme, we proposed a dynamic skew-and-split tree, DISSECT, which progressively adjust the tree height by varying the tree arity to reduce authentication overhead. DISSECT can mitigate the frequent overflowing of counters that are used for encryption when the tree arity is increased. Experimental results show an average of 20% reduction in authentication time over the balanced tree approach. Finally, we evaluated the benefits of integrating a cryptographic hardware module in the memory controller. This resulted in a 40-50% decrease in authentication time during memory authentication.

Contents

Statement of Originality	iii
Supervisor Declaration Statement	v
Authorship Attribution Statement	vii
Acknowledgments	xi
Abstract	xiii
List of Figures	xix
List of Tables	xxi
List of Algorithms	xxii
1 Introduction	1
1.1 The Need for Secure Embedded Systems	1
1.2 Securing Memories in Embedded Systems	3
1.2.1 Technical Challenges	4
1.3 Objectives	7
1.4 Main Contributions	8
1.5 Publications	10
1.6 Thesis Organization	11
2 Literature Review	13
2.1 Securing Embedded Systems	13
2.1.1 Software Attacks	15
2.1.2 Side-Channel Attacks	18
2.1.3 Physical Attacks	19
2.1.4 Shortcomings of Existing Works	20
2.2 Memory Attacks and Protection Schemes	21

2.2.1	Types of Memory Attacks	22
2.2.2	Countermeasures	23
2.3	Discussion	27
2.3.1	Limitations of Existing Memory Authentication Schemes	27
2.3.2	Opportunities for Reducing Authentication Overhead	28
2.4	Summary	31
3	Static Skewed Memory Integrity Tree	33
3.1	System Overview	34
3.1.1	Threat Model	34
3.1.2	Security Model	34
3.2	TEC-tree	36
3.3	Proposed Method	38
3.3.1	Static Skewed Memory Integrity Tree Model	38
3.3.2	Static Skewing Algorithm	41
3.4	Implementation Example	46
3.5	Performance Evaluation	49
3.6	Summary	50
4	Dynamic Skewed Memory Integrity Tree	53
4.1	System Overview	54
4.2	Proposed Method	55
4.2.1	Dynamic Skewed Memory Integrity Tree Model	55
4.2.2	Dynamic Skewing Algorithm	57
4.3	Implementation Example	62
4.4	Performance Evaluation	63
4.5	Summary	64
5	Architecture-Aware Dynamic Skewed Memory Integrity Tree	67
5.1	Overview of Proposed Security Model	68
5.2	Proposed Framework	70
5.2.1	Cache-Aware Memory Integrity Tree Model	71

5.2.2	Authentication	74
5.3	Implementation Example	74
5.4	Performance Evaluation	76
5.4.1	Authentication Time	77
5.4.2	Energy	79
5.4.3	Instruction Per Cycle (IPC)	82
5.5	System Overhead	84
5.6	Summary	86
6	Performance Optimizations and Security Analysis	87
6.1	Dynamic Skewed Memory Integrity Tree with Tree-Cache	87
6.1.1	System Overview	88
6.1.2	Proposed CADST Framework	90
6.1.3	Proposed Skewing Framework	91
6.1.4	Performance Evaluation	95
6.2	DISSECT: Dynamic Skew-and-Split Tree	97
6.2.1	Proposed Tree Structure for DISSECT	102
6.2.2	Proposed Skew-and-Split Framework	103
6.2.3	Performance Evaluation	107
6.3	Custom Instructions	109
6.3.1	AES Algorithm	109
6.3.2	Performance Evaluation	113
6.4	Security Analysis	114
6.5	Summary	120
7	Conclusion and Future Work	123
7.1	Conclusion	123
7.2	Future Work	125
7.2.1	Energy-Efficient Secure Memory Controller	126
7.2.2	Memory Authentication in Hosted Environments	126
7.2.3	Dynamic Integrity Tree for Multi-Core Platforms	126
7.2.4	Mitigating Reverse Engineering Attacks	127

List of Figures

1.1	Embedded system market size, by application, 2012-2023 (USD Billion) [6]	2
2.1	Three different kind of attacks: (a) Splicing, (b) Spoofing, and (c) Replay [7]	24
2.2	Fully balanced memory integrity tree	29
2.3	Skewed memory integrity tree	29
3.1	Security design overview	35
3.2	A TEC-tree for 8 Data Chunk blocks and 4 levels	36
3.3	A sample Data Chunk	37
3.4	A sample Counter Chunk	37
3.5	Example of a modified TEC-tree	39
3.6	Package-Merge Algorithm	47
3.7	Calculating levels	48
3.8	Performance gain in comparison to a balanced tree	50
4.1	System design overview	54
4.2	Example of a dynamic tree structure	56
4.3	Shifting up of sub-tree C: (a) Exchanging with uncle A, (b) Rotation, (c) Final state	60
4.4	Tree migration and rebalancing steps	62
4.5	Performance evaluation: (a) Number of tree levels accessed, (b) Time overhead for dynamic tree, and (c) Authentication time	66
5.1	Security design overview	69
5.2	Proposed dynamic skewed integrity tree structure	73
5.3	Implementation example	75

5.4	Authentication time (in ms) for twelve different applications with varying cache configurations	81
5.5	Instruction per cycle comparison	84
6.1	Security design overview	89
6.2	Standard memory integrity tree structure	91
6.3	Proposed vEB memory integrity tree structure	92
6.4	Skewed tree with sub-trees of size less than CL	93
6.5	Authentication time with a) 16KB Tree-Cache, 64B cache line, b) 16KB Tree-Cache, 128B cache line, c) 8KB Tree-Cache, 64B cache line, d) 8KB Tree-Cache, 128B cache line	100
6.6	CCs per DC accessed	101
6.7	An example structure of (a) 4-ary DISSECT, (b) Post skew, (c) Post skew-and-split	101
6.8	(a) Proposed node structure, (b) Memory block with 512-bit cache line for TEC-tree, and (c) Memory block of DISSECT	103
6.9	Analysis of counter overflow problem	107
6.10	Normalized authentication time	108
6.11	Flowchart for a generic AES algorithm for encryption	110
6.12	Key Expansion	111
6.13	Substitution Byte	112
6.14	Shift Row	112
6.15	Mix Column	112
6.16	Authentication time (in ms) for twelve different applications with varying cache configurations	116

List of Tables

1.1	Reported attacks on embedded systems	5
2.1	Attacks on embedded systems	14
3.1	Time taken to create static skewed tree	49
4.1	Lookup table	57
5.1	Memory access patterns	78
5.2	4MB Balanced vs 1MB Proposed	83
5.3	1MB Balanced vs 256KB Proposed	83
6.1	Architectural parameters used in simulations	95
6.2	Height of memory integrity trees	97
6.3	Security limitations	118

List of Algorithms

3.1	Package-Merge (I, X)	44
4.1	Dynamic Skewed Tree	58
4.2	Set Migration $\{a : \textit{data element}\}$	61
4.3	ShiftUp $\{T : \textit{pointer to node}\}$	62
6.1	Dynamic Splitting	106

Chapter 1

Introduction

The pervasiveness of embedded systems in our daily lives makes security a key design objective. Recent foray of security features in commercial embedded processors reflect the growing security concerns. However these security features cannot extend fully beyond the processor core to guarantee secure storage in external third party memories, therefore exposing the vulnerability of the system to memory attacks. This presents a growing threat to cyber-security with the increased used of third-party IP (3PIP) cores and storage elements in embedded systems. Furthermore the tight computational constraints imposed on embedded devices present a significant challenge for adopting strong memory security schemes which are computationally expensive.

In this chapter, we first introduce the need for securing embedded systems. In particular, we will discuss the memory security concerns and the challenges of memory protection in embedded systems. We will list the major research objectives and our significant contributions in this thesis to address these challenges. Finally, we provide the organization of the thesis.

1.1 The Need for Secure Embedded Systems

Embedded systems have become an integral part of our everyday lives. Fig 1.1 shows a clear trend of the embedded systems market growth in different sectors, which is expected

to exceed USD 258.72 billion by 2023. We use these devices on a daily basis to process and store sensitive information in healthcare devices, for military purposes, etc., and hence, they must be secure. The functionalities of embedded systems are often enhanced by providing them with networking capabilities. This has led to the Internet-of-Things (IoT) paradigm, which is essentially a network of embedded devices that are enabled with sensing capabilities and can interact with multiple entities to relay information about their environments. However, the networking capability of the embedded systems also lead to major security concerns. Once attacked, an IoT device will function as a pivot point for adversaries to move laterally across the network and exploit other devices, causing detrimental effects on a large user base [8].

Figure 1.1: Embedded system market size, by application, 2012-2023 (USD Billion) [6]

With the globalization of the semiconductor industry, the market demands for intricate embedded system design have soared. Companies have started to outsource various aspects of design and fabrication. The increasing adoption of 3PIP cores and storage elements in embedded systems [9] allow for far more devastating effects of cyber-crime than has been

reported so far. As reported in [10], attacks on embedded systems have been steadily increasing for almost two decades now. Healthcare data collected from wearables are reportedly up to 10 times more valuable than credit card data [11]. In 2014, cyber criminals compromised approximately 4.5 million patient records [12]. Successful cyber-attacks on implantable/wearable devices have already been demonstrated [13–16]. Hardware-oriented attacks are also becoming a serious concern as gaining access to these devices is relatively simpler. Targeted attacks through data modification can cause devastating outcomes on the user or on a large user base. A recent report suggests that costs of cyber-attacks are USD 400 billion per year [17]. The cost of a single cyber-crime is averaged at USD 12.7 million. It takes 259 days on average to detect an attack and 45 days on average to fully recover from it [18]. It is likely that for small and medium-sized enterprises, the cost of recovery will be permanently damaging.

The main motives of typical attacks on embedded systems are retrieval, use, disclosure, disruption, modification, or destruction of sensitive information or information systems. As such, an embedded system should ideally incorporate the following security requirements [19]:

- **Confidentiality:** Protect sensitive data from being exposed.
- **Integrity:** Ensure that any data is not tampered.
- **Availability:** Perform the correct functionality when required.

1.2 Securing Memories in Embedded Systems

Table 1.1 lists some reported attacks on embedded systems, their impact and the system component that is targeted by the attack. It can be observed that memories are one of the main target areas of attacks. Failure to protect memories makes data transfers easily observable and enables the leaking of sensitive information in the process. Memories are

also vulnerable as they are hard to audit and rarely visible to the software. Although, many memory encryption techniques have been developed, the recent increase in the sophistication of attacks has rendered simple encryption schemes incapable of protecting memories [20]. An adversary can obtain nontrivial information about the execution of a program and the nature of the data that it is dealing with, just by observing the pattern in which various locations of memory are accessed during its execution.

Commercial processor-level countermeasures typically define notions of trust zones or boundaries (or security perimeters) across the various on-chip hardware resources [21–24]. Oblivious RAM (ORAM) [25] algorithms allow a client to conceal its access pattern to the remote storage by continuously shuffling and re-encrypting data as they are accessed. However the protection schemes in current trusted computing platforms cannot extend fully beyond the processor core to guarantee secure data storage in external third party memories [26] without incurring large runtime overhead. This is a significant concern as the motive of almost all attacks is exploiting leakage or modification of information, making memory an obvious target. Hence, confidentiality and integrity of memory data in embedded systems have been a longstanding issue in trusted system design [27].

1.2.1 Technical Challenges

Embedded systems differ from conventional computer systems in several ways, making it more difficult for them to protect against security attacks. The challenges for protecting embedded systems against attacks stem from the following reasons [28].

- **Limited Computational and Storage Capabilities:** Due to size and cost requirements, embedded systems are generally equipped with processors with limited processing power, and memories with low storage capacities. At the same time, the embedded applications may need to meet real-time performance. As a result, the available computational and storage resources cannot accommodate to the

Table 1.1: Reported attacks on embedded systems

Attack	Impact	Target	Reference
Stuxnet	Targets industrial computer systems and was responsible for causing substantial damage to Iran's nuclear program	<i>Software</i>	Industrial Electronics Society, IEEE, 2011
Sony PlayStation	Lost \$1 billion, started as a glitch attack that allowed the hacker to enter a privileged processor	<i>Software</i>	PlayStation 3 Hacked BBC News 25, 2010
Denial of Service	A reported mobile phone attack that prevented users from making a 911 emergency call	<i>Availability</i>	Embedded System Security. Applied Data Systems, USA, 2004
German Power Company	Lack of memory encryption in a smart electric meter design led to an estimated \$400 million loss	<i>Memory</i>	Electronic Design, 2016
Heartbleed	Remote reading of RAM working memory in an estimated half million secure web servers worldwide	<i>Memory</i>	Internet Measurement, ACM, 2014
Satellite Attack	Take control of a British military communications satellite	<i>Availability, Memory</i>	Xnet, 2007
Pace Makers	Pacemakers and ICDs were hacked to interpret private data transmissions	<i>Memory</i>	Security and Privacy, IEEE, 2008
Supermicro	Tiny network monitoring and control chips were implanted on motherboards of white-box servers, used by companies like Amazon and Apple	<i>Side Channel</i>	Bloomberg Business Week, 2018
Nest Thermostat	Attacker uses a global reset flaw followed by a USB stick connection to completely alter the behaviour of a smart home automation device	<i>Software</i>	Black Hat, 2014

demands of compute and memory intensive security schemes. For example, existing memory authentication techniques have reported a storage overhead that ranges from 300-1000 bytes per packet, while the total memory capacity of the system is only 8KB [29].

- **Limited Power Budget:** Many embedded systems are battery operated and hence they have tight power budgets. Such systems are unable to run the standard defense procedures for computer systems such as desktops, workstations and servers, as they have limited resources that can be dedicated to the security primitives and protocols. For example, the RSA (Rivest, Shamir, and Adelman) algorithm used to encrypt 1024 bits of data for secure transmission will consume 42mJ of energy on a Motorola DragonBall MC68328 processor, which far exceeds a typical battery capacity of only 26KJ [30].
- **Physical Exposure:** Embedded systems are generally not kept in a close and safe environment, but rather they are easily accessible to the adversaries, which increases the risk of physical attacks. Some embedded systems are even employed in remote and harsh locations making it difficult for security patch updates.

It is worth noting that an attacker will generally use a combination of attacks to achieve his purpose. Attackers are constantly devising ingenious schemes and improvising on previous attacks to compromise the system's security. Although a large body of research has been undertaken to detect the presence of attacks, there is still no single comprehensive solution. This challenge is compounded by the fact that protecting embedded devices from memory attacks cannot violate the tight performance constraints imposed on the systems.

In order to mitigate the security threats on memories, secure architectures have been proposed with the underlying assumption that the processor chip is secure and

off-chip devices (e.g. memories) are insecure [31,32]. To counter physical attacks, all sensitive information that needs to be transmitted from the secure processor (e.g. to external memory) are encrypted [33]. Although a number of memory encryption schemes have been proposed [34–36], they entail significant performance overhead (up to 25% performance overhead) [37]. Protection from replay, spoofing and splicing attacks can be achieved through memory authentication schemes that perform runtime integrity verification [7]. The integrity of the code and data in memory is protected by maintaining memory integrity trees in order to preserve the state of the application. However, these security mechanisms are often computationally intensive and account for excessive memory access [37]. As memory accesses of code/data contribute to the largest fraction of energy consumed in embedded processors [38], such methods are often infeasible for embedded systems with tight functional constraints. Therefore, runtime integrity checking has been identified as one of the main open issues for trusted computing in embedded systems [39].

1.3 Objectives

The main aim of this research is to reduce the performance overhead for securing untrusted memories in embedded systems. We will focus on securing memories from active memory attacks (e.g. replay, spoofing and splicing attacks) that can compromise the confidentiality and integrity of the memory data. It is evident that the biggest challenge in protecting embedded systems from malicious attacks lie in the tight functional constraints imposed on them. While a number of countermeasures exist, they often incur significant performance overheads. Our hypothesis is that these overheads can be effectively mitigated by customizing the memory integrity tree based on the runtime memory access patterns. We will propose architecture-aware algorithms to support the adaptation of memory integrity trees to application workloads in order to reduce the runtime overheads without compromising on the security strength. The objectives of this research are to:

- Investigate compile-time techniques to generate performance-optimized memory integrity tree structure for any given application based on their static application characteristics.
- Propose strategies to dynamically restructure the memory integrity tree based on the runtime memory access patterns.
- Develop a framework that combines the dynamic memory integrity tree approach with methods for customizing the integrity tree structure based on the cache configuration in order to minimize the performance and energy overhead through speculative authentication.
- Devise optimization schemes that lead to scalable dynamic memory integrity tree solutions for applications with large number of memory accesses.

1.4 Main Contributions

The major research contributions of this thesis are summarized as follows:

- (i) We proposed a method to construct customized memory integrity trees by utilizing the package-merge algorithm to generate a skewed memory integrity tree based on the static analysis of memory data accesses. To the best of our knowledge, this work is the first to propose an automated approach for generating customized memory integrity trees. To demonstrate the effectiveness of the approach, we implemented the proposed method on the Altera NIOS II processor with an external processor DRAM to perform runtime memory integrity verification. This work has been published in [1].
- (ii) We developed an efficient technique to dynamically customize the memory integrity tree based on the runtime memory access patterns to reduce the authentication overhead. The proposed method can restructure the memory integrity tree at

runtime such that the nodes that correspond to frequently accessed data are placed closer to the root. We validated the effectiveness of our approach on the Altera NIOS II processor with an external DRAM. This work has been published in [2].

- (iii) We proposed a framework that combines architecture-specific customization of the integrity tree with runtime mechanisms for accelerating the authentication process. The former tailors the leaf node structure of the tree based on the cache configuration of the target processor in order to take advantage of speculative authentication and also to maintain cache line granularity of memory accesses. At runtime, the tree is dynamically skewed such that frequently accessed data are moved closer to the root. The framework is simulated with Multi2Sim and compared with other existing mechanisms to demonstrate its performance and energy benefits. This work has been published in [3].
- (iv) We investigated the benefits of employing a dedicated Tree-Cache for caching the memory integrity tree nodes as the tree is dynamically restructured. The tree layout is initialized with a van-Emde Boas (vEB) organization to reduce the memory-cache accesses and the skewing algorithm shifts memory blocks to form sub-trees in order to maintain minimal number of cache misses during tree traversal. Experimental results demonstrate a speedup of close to 10% over a system with a standard Tree-Cache and nearly 50% over a system without Tree-Cache.
- (v) We propose a framework called DISSECT (Dynamic Skew-and-Split Tree for Memory Authentication), which enables the height and arity of the memory integrity tree structure to be dynamically adjusted based on the runtime memory access patterns in order to reduce the performance overhead for authentication. DISSECT overcomes the limitations of existing methods by mitigating the negative effects of frequent

overflowing of counters that are used for encryption in the tree when the arity is increased. This work has been published in [4].

- (vi) Finally, we quantify the effects of employing the AES algorithm as a custom instruction for memory authentication. The results show that significant performance gains can be achieved by employing a custom hardware cryptographic module that is tightly coupled with the memory controller for memory authentication.

1.5 Publications

The following lists the publications by the author based on the research outcomes of this thesis:

International Refereed Journal

- (i) Vig, Saru, Rohan Juneja, Guiyuan Jiang, Siew-Kei Lam, and Changhai Ou. "Framework for fast memory authentication using dynamically skewed integrity tree." IEEE Transactions on Very Large Scale Integration (VLSI) Systems 27, no. 10 (2019): 2331-2343.

International Refereed Conferences

- (i) Vig, Saru, Tan Yng Tzer, Guiyuan Jiang, and Siew-Kei Lam. "Customizing skewed trees for fast memory integrity verification in embedded systems." 2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), pp. 213-218. IEEE, 2017.
- (ii) Vig, Saru, Guiyuan Jiang, and Siew-Kei Lam. "Dynamic skewed tree for fast memory integrity verification." 2018 Design, Automation, and Test in Europe Conference & Exhibition (DATE). IEEE, 2018.

- (iii) Vig, Saru, Sarani Bhattacharya, Debdeep Mukhopadhyay, and Siew-Kei Lam. "Rapid detection of Rowhammer attacks using dynamic skewed hash tree." Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy (HASP), p. 7. ACM, 2018.
- (iv) Vig, Saru, Rohan Juneja, and Siew-Kei Lam. "DISSECT: Dynamic Skew-and-Split Tree for memory authentication." 2020 Design, Automation and Test in Europe Conference & Exhibition (DATE). IEEE, 2020.

1.6 Thesis Organization

The thesis is organized as follows:

- Chapter 2 reviews the related work in the literature on attacks and countermeasures with respect to embedded systems security. We will discuss the security attacks on memories in embedded systems and limitations of existing countermeasures.
- Chapter 3 describes the proposed approach for constructing a static skewed memory integrity tree for memory protection. We will provide detail description of the package-merge algorithm for customizing the skewed memory integrity tree and the evaluations on a Field Programmable Gate Array (FPGA) system.
- Chapter 4 introduces the proposed method to dynamically reconstruct the integrity tree based on runtime memory access frequencies. We will present experimental results on a FPGA system to demonstrate the benefits of the proposed method.
- Chapter 5 presents our framework for customizing the dynamic skewed memory integrity tree based on the cache configuration of the system. We also discuss the impact of various cache configurations on the performance and energy overhead of the authentication method.

- Chapter 6 discusses three optimization techniques for increasing the scalability of our proposed dynamic integrity tree approach. These optimization techniques include the effective use of a dedicated Tree-Cache, a framework that can adjust the arity of the tree dynamically while reducing counter overflows that negatively impact the performance of memory authentication, and the utilization of custom instructions to accelerate the encryption and decryption steps during memory authentication. The security analysis for the proposed dynamic skewed memory integrity tree is also presented in this chapter.
- Chapter 7 concludes the thesis and discusses the planned future directions of this research.

Chapter 2

Literature Review

In this chapter, we provide a detailed survey of the existing attacks on embedded systems and the prevailing protection techniques. We will discuss the limitations of the current works and highlight why the external memory of embedded systems is a target of numerous attacks. The chapter concludes with a discussion of the critical observations and the research challenges based on this survey, leading to the research contributions in the chapters that follow.

2.1 Securing Embedded Systems

Security attacks on embedded systems can be categorized as follows based on the various ways that they can be launched:

- **Software Agents:** Attacks can be launched through software agents like virus, trojan horses, worms etc.
- **Side-Channel Probing:** Attackers can extract the secret information (such as a cryptographic key) from a victim program by observing a physical phenomenon, e.g. execution time and power consumption, during its execution.
- **Physical Intrusion:** These are invasive attacks that require physical intrusion at the chip, board, or system level.

Table 2.1: Attacks on embedded systems

Classification	Example Attacks	Brief Description
Software	Buffer Overflow	Overflow data to adjacent memory locations [40].
	Virus	Alter the normal program execution on the computer [41].
	Code-Reuse	Manipulate program control flow [42].
	Data-flow	Corrupt memory load and store operations [43].
Side-Channel	Timing	Analyze execution time to identify data-cryptographic computations [44].
	Power	Analyze power profile of cryptographic computations to reveal secret key [45].
	Electromagnetic	Measure the electromagnetic radiation emitted by a device to reveal sensitive information [46].
	Cold-boot	Access memory dump of the random access memory [47].
Physical	Optical fault injection	faults into system using focused camera flash or laser pointer [48].
	FIB editing	Modify chips to obtain, leak or corrupt [49].
	Fault attack	Induce faults through external parameters and environmental conditions of a system such as the supply voltage, clock, temperature, radiation, etc. [50].
	UV light	Disable security fuses in EPROM [51].
	Laser attack	Split chip into several slices to analyze its layout [52].

Table 2.1 provides examples of attacks and their brief description based on the above-mentioned classification. In practice, attackers often use a combination of techniques to achieve their objectives. The following sub-sections provide a brief introduction to each of these class of attacks, followed by a discussion on existing protection schemes employed against them.

2.1.1 Software Attacks

Software poses a significant security threat in embedded systems. The evolving expertise and finesse of the attackers continue to exploit the vulnerabilities of the software despite the increasing complexity of software applications [53]. Moreover, with embedded systems being increasingly interconnected with IoT, even a small software vulnerability can cause a significant security breach. The biggest challenge in preventing a software attack stems from the fact that the physical presence of an attacker is not necessary for mounting the attack. In a typical software attack, like code injection, the attacker can inject malicious code remotely through the internet [28]. Buffer overflow attacks occur when data in a memory buffer exceeds the allocated size, causing data to overflow into adjacent storage. An attacker could exploit this overflow to gain access to a system [40]. A computer virus is a type of malicious code or program written to alter the way a computer operates and is designed to spread from one computer to another. In a code-reuse attack, upon encountering the corrupted code pointer, the program control flow is diverted to a part of the existing code [42]. An attacker can re-purpose the existing code to perform arbitrary computations. It is also possible to mount pure data-oriented programming (DOP) attacks which only corrupt memory load and store operations, without inducing any unintended program flows [43]. Due to the vast nature of the problem and a steady increase in the expertise of attackers, there is still no comprehensive solution to protect against all kinds of software attacks. In the following section, we discuss some of the prevalent protection schemes against software attacks.

2.1.1.1 Attack Resistant Architectures

System architectures need to be modified to make them more resistant and conducive for executing security mechanisms. Palladium [54] is one such initiative by IBM to provide secure data transfer for large content providers over the internet. It has now been expanded to cover other security threats as well and goes by the name of Next Generation Computing Base (NGSCB) [55]. Trusted Computing Group [32] is another not-for-profit group which works towards proving hardware enhancements for secure processing. An important criterion for these modified systems is the backward compatibility with existing non-secure operating systems. The key features expected from any trusted computing base are listed below:

- **Memory Curtaining:** The sensitive areas of the memory are isolated, and any inappropriate reading or writing should be detected and avoided.
- **Secure Input and Output:** To tackle threats such as key-loggers (a spy attack where the keypresses on the keyboard are recorded without the user's knowledge), and programs that can capture the screen display.
- **Sealed Storage:** Securely saves sensitive information such as cryptography keys by binding it to the platform. Essentially, only a particular software and hardware combination can release the data.
- **Remote Attestation:** This enables the system to detect any unauthorized changes to the software remotely. Trusted computing architectures generates encrypted certificates attest to a remote party residing outside the trusted system.

Several processors have been developed to provide a subset of the features stated above. ARM has announced a Trust Zone Technology [24] targeting the system-on-chip market. Smart-MIPS [56] developed by MIPS has specific memory protection and encryption features.

2.1.1.2 Secure Bootstrapping

The work in [57] discusses a layered boot-up process where the booting transitions require an integrity check between each layer, before handing over the reins to the operating system. Hash values based on the boot-up component are calculated and compared to a secured value.

2.1.1.3 Operating System based Countermeasures

The work in [58] implements a split memory process. The memory is split into at least two different sections for code and data. The code segment will be made read-only and the data segment non-executable [59,60]. Other proposed enhancements to the OS are context switching, exception handling, memory management, and inter-process communication. System architectures need flexibility to incorporate such changes [61,62].

2.1.1.4 Software Authentication

Basic techniques for verifying software programs before execution are checksum or hashing. In [63], authors introduce a novel method of oblivious hashing for runtime verification. Sand-boxing [64] is another approach based on allocating restricted environments to the code. The compiler bundles security proofs along with the program executables. The program itself should be able to generate evidence of maintaining the consistency of the kernel's data structures.

In [65], the authors suggest prior recording of all patterns of normal behavior and comparing them with executed system calls. Any deviation will raise the alarm. However, any update made to the application post-recording of the regular routine will trigger a false positive alert.

2.1.2 Side-Channel Attacks

In side-channel attacks, the adversary attempts to analyze any repeatable patterns in the timing, power, and electromagnetic traces. Such traces can be leaked through various channels during computations. A successful side-channel attack can lead to deciphering the encryption keys by statistical analysis. These attacks are commonly reported on smaller devices such as smart-cards, Personal Digital Assistants (PDAs), and mobile phones [61,66]. A cold boot attack (or to a lesser extent, a platform reset attack) is a type of side-channel attack in which an attacker with physical access to a computer performs a memory dump of a computer's random access memory by performing a hard reset of the target machine [47].

2.1.2.1 Protection Against Side-Channel Attacks

Randomization is frequently used as an effective countermeasure against any side-channel attack that requires the attacker to know exactly when a certain operation is performed [67]. It can be randomization of the execution sequence or the addition of a random power dissipation module or a current sink [68]. This prevents the attacker from extracting any meaningful timing or power information from the processor. Another technique is data masking, where noise is added to mask the code execution to confuse the adversary. Some architectures employ the window method, i.e., the cryptography key is calculated over a specific number of iterations per window, and the window size is chosen randomly. [69] propose a software-based approach of code balancing. Complementary coding events shield actual computations, and dummy instructions can also be placed at random locations to increase noise in the leaked signals. [70] has a non-deterministic processor design where independent instructions are executed in an out-of-order random fashion. Authors in [71] discuss the re-distribution of on-chip components to break any locality properties thereby preventing electromagnetic leakage.

2.1.3 Physical Attacks

Physical attacks are invasive in nature, i.e., the attacker will need to gain access to the system. They require expensive equipment, knowledgeable attackers, and sometimes significant amount of time. Equipment utilized for physical attacks include high-resolution optical microscope, laser cutting system, and micro-probing station. UV light has also been used to disable security fuses in EPROM [51]. Focused ion beam technology (FIB) has been employed for chip modification to obtain, leak or corrupt sensitive data, security keys, or confidential information [49]. Probes have also been used for eavesdropping by analyzing the bus transactions and data transfers between the processor and memory [34].

2.1.3.1 Encryption Schemes

Authors in [33] discuss different memory encryption schemes for secure data transmission. Conventionally, a symmetric key cipher is employed to encrypt data to meet the primary aims of confidentiality and integrity. But ciphers can also be designed using asymmetric, or hash algorithms. Hardware acceleration of such algorithms can reduce performance overhead and energy consumption of the system as ciphering is generally a computationally intensive task. Due to the high implementation cost of a pure hardware solution, a combination of a hardware-software is more feasible. For example, to implement the RSA algorithm, the hardware accelerator can be used to perform modular multiplication, while the rest of the algorithm is executed in software. Another such approach is through the use of custom instructions [35], where the hardware cryptographic accelerator is tightly integrated with the processor core.

2.1.3.2 Memory Authentication Schemes

Typical memory authentication schemes employ an integrity tree to perform runtime integrity verification [37]. Authentication methods like hash function, Message authentication code (MAC), Block-level AREA are used to realize the integrity trees [34, 72]. The

root value of the tree is stored on-chip and is assumed to be safe and resistant to tampering. Existing tree techniques include HASH trees, Parallelizable authentication (PAT) trees, and Temper evident counter (TEC)-tree [7]. However, implementing such integrity trees significantly increases runtime and memory overhead [37, 73], making them infeasible for embedded systems with tight energy constraints. Therefore, runtime integrity checking has been identified as one of the leading open issues for trusted computing in mobile devices [39]. Other works include a monitoring unit to verify the integrity of instructions before committing them [74].

Intel’s SGX [36] employs an integrity tree as part of its memory encryption engine with counter mode encryption and MAC as its primitive. PoisonIvy has a similar architecture, but uses speculative execution of instructions to improve its performance [75]. It tracks data/addresses sent to the memory speculatively to ensure that no unverified data escapes the chip. Merkle integrity trees in [76] process data in batches to reduce the on-chip storage.

2.1.4 Shortcomings of Existing Works

System security is an active area of research, where many ongoing efforts are being undertaken to develop new attacks and defenses. There is a large body of research in software and side-channel attacks, where current works focus on general computer systems as well as embedded systems. Despite the fact that memory integrity is considered as one of the leading open issues in trusted computing for embedded devices, there is relatively lesser research efforts in this area. This will become increasingly important with the ease of accessibility of embedded devices for physical attacks on memories.

Commercial processor-level security countermeasures typically define notions of trust zones or boundaries (or security perimeters) across the various on-chip hardware resources [24] [77]. However, the protection schemes in current trusted computing platforms cannot

extend fully beyond the processor core to guarantee secure code/data storage in external third party memories [26]. This enables adversaries to mount active attacks on memories such as cold-boot attacks by exploiting memory remanence properties [78] (to extract confidential information), or perform replay, spoofing and splicing attack [37] (to influence program execution or reveal data by changing memory contents). Moreover, existing disk encryption techniques support encryption of data at rest, which is less useful for mobile embedded systems that can be easily hijacked by adversaries [79]. Hence, the security and integrity of memory in embedded systems have been a long-standing issue in trusted system design [27].

There is still a wide gap in knowledge and techniques to reduce the steep cost of memory authentication schemes in terms of performance overhead, energy consumption, and memory storage overhead. Secure processor architectures could surely benefit from a more robust protection scheme for external memories which do not incur significant runtime overhead. In the next section, we will provide a detailed discussion on memory attacks and existing memory-protection systems, with emphasis on runtime integrity verification.

2.2 Memory Attacks and Protection Schemes

Main memory is a critical component of embedded systems that needs to be secured. Memory is hard to audit, rarely visible to the software, and is relatively an easier target for attacks. Successful memory security measures involve physical protection (to ensure memory hardware and media are not stolen or damaged), minimizing the risk and implications of error, failure or loss (e.g. developing a resilient back-up strategy), applying appropriate user authentication (e.g. employing secure password scheme), as well as implementing the encryption of sensitive data and/or cryptographic schemes for memory authentication. Secure architectures have been proposed to mitigate the security threats

on memories. The underlying assumption of these architectures is that the processor chip is safe, and off-chip devices (e.g., memories) are insecure [80].

The following sub-sections discuss the attacks on the main memory and the prevalent protection mechanisms along with their drawbacks.

2.2.1 Types of Memory Attacks

The following provides a list of some known memory attacks [81]:

- **Cold Boot Attacks:** These attacks are based on the data remanence property of DRAMs. The DRAM memory cells are still readable for a few minutes after the system power is cut off. Attackers can exploit this phenomenon to extract information from the memory [82].
- **Side-Channel Attacks:** Such attacks aim to infer memory bus transactions while the system is in operation by analyzing the power and timing traces collected from the side-channel leakages. By employing statistical analysis on the power and timing signals, the attacker can decipher the cryptographic keys used to encrypt the data, hence compromising the data confidentiality [83].
- **RowHammer Attacks:** RowHammering exploits the interference properties of DRAM cells due to high bank locality accesses. By repeatedly opening and closing DRAM rows from the same bank within one refresh cycle, the bits in the DRAM memory cells may flip [84].
- **DMA Attacks:** An attacker could program a DMA-capable peripheral to manipulate the DMA controller and read arbitrary memory regions. This class of attacks is often referred to as *Firewire attacks* [85].
- **Code-injection Attacks:** This class of attacks inject or modify code running on the platform. Attacker can use a compromised DMA, or attach a logic analyzer

to the memory. These attacks are very challenging to mount because they are electrically unsound [86].

- **Bus Attacks:** These attacks replace the main memory accesses with some malicious content. The typical bus attacks are shown in Fig. 2.1. In particular, they can be classified into the following:
 - **Spoofing:** Attacker exchanges a memory block with a tampered one.
 - **Splicing:** Attacker replaces the memory block at address A with a memory block B where $A \neq B$.
 - **Replay Attacks:** Attacker records data at an address and inserts it at the same address at a later point in time. Thus, the present value of the data is replaced by an older value.

2.2.2 Countermeasures

Architecture modifications proposed in [57,87] require a separate core on-chip to handle authentication schemes. Instruction and data accesses are monitored, and the security core can be customized as per application and security needs. The authors in [88] introduce memory heat map (MHP) behavior; a machine-learning algorithm is employed to detect anomalies in the memory behavior patterns. An on-chip memometer monitors all memory activities. The authors in [89] propose a technique to randomize the memory space and also insert a watchdog timer to monitor all activity. [90] design a new cache architecture to thwart memory side-channel attacks. A novel random replacement algorithm which can detect attacks without degrading performance is employed.

Several memory integrity check schemes have been proposed to enable user authentication. [91] present a one-time password (OTP) and a Cyclic Redundancy Check (CRC) module to secure the integrity of external memory. Before OTP generation, they generate

Figure 2.1: Three different kind of attacks: (a) Splicing, (b) Spoofing, and (c) Replay [7]

CRC for the cache line to be encrypted and store it in cache. During verification, the CRC is re-generated and compared with the stored CRC. [92] propose a light-weight and cost-effective tag generation system for authenticating off-chip memory. [93, 94] secure the memory address buses in addition to user authentication. This prevents the memory access patterns from being revealed to the attacker. Such schemes, although successful in detecting attacks, have excessive memory overhead due to the storage of memory authentication primitives, e.g., CRC, tags. To reduce this excessive storage, many schemes have been proposed that rely on integrity trees.

2.2.2.1 Memory Integrity Trees

Typical techniques to protect memory inevitably use some form of encryption and a memory integrity tree. Integrity tree-based techniques (see example Fig. 2.2) split the memory into equal-sized blocks, $Data_i$, where $i = 1, 2, \dots, n$ is the node number on the tree. Data blocks form the leaf nodes of an A -ary tree. The remaining nodes are created by recursively applying a primitive authentication function over the memory blocks and are called counter blocks, denoted by CC_i . Recursion is performed until a single root node CC_1 is obtained. The root CC_1 is securely stored on-chip, and it represents the present state of the system. Thus any tampering can be detected at runtime if the root value does not match the value stored on-chip. In the following, we will discuss the various memory integrity tree models.

Merkle Tree Merkle tree was devised by Merkle [95] and has been documented as the first integrity tree used for memory authentication using cryptography. Merkle tree uses a simple hash function, $H()$, as their primitive authentication technique. Thus, nodes of the tree are hash values. Prior to the introduction of Merkle tree, the strategies adopted was to store on-chip a hash value for every memory block that is stored off-chip. For each memory access operation, the hash value of the corresponding memory block is computed and compared to the one stored on-chip. Any bus tampering attack is detected if there is a mismatch between the hash values. The Merkle tree structure was introduced to reduce the memory cost of storing all the hash values. Nodes of the tree are recursively hashed till a single hash value (the root hash) is obtained, which is then stored on-chip. The root value stored on-chip is assumed to be tamper-resistant and inaccessible to adversaries. The read operations can be performed in parallel with the authentication as the required hash values can be made available at the very start, but for write operations, this is not possible. A node will have to wait for its children nodes to be updated first before their hash values can be calculated and stored.

PAT Tree PAT tree [96] uses MAC function instead of hash for authentication. Each memory block has a unique MAC value that can be computed using a key. The key is stored on-chip in an tamper-resistant area. Thus, MAC values can be stored off-chip as a valid MAC will only be generated using the key stored on-chip. To ensure that each location has a unique MAC, the pre-image of the MAC stores a nonce (number used only once). On a read operation, the memory block, along with its MAC, are loaded onto the processor. To check for its validity, MAC is re-computed on-chip, using the key and a copy of the nonce. There is a significant memory overhead for storing the nonces on-chip to re-generate the MAC values. The PAT tree uses the MAC and reduces the memory overhead by storing only the root nonce and key on-chip. To verify the integrity of a memory block, the nodes are recursively verified at all levels of the tree until the root. If the MAC matches with the one generated using the on-chip nonce and key, the data is considered safe and can be sent to the processor.

Tamper Evident Counter Tree The Tamper-Evident Counter Tree (TEC-tree) [33] uses a primitive known as Block-level AREA, which combines both encryption and authentication into a single operation. A nonce, consisting of the address and a counter, is generated and concatenated onto every data block (cache line) and the resulting block, called the Data Chunk (DC), as shown in Fig. 3.3, is then encrypted with a symmetric key cipher operating in Electronic Code Book Mode (ECB). The nonce are combined to form more data blocks, called the Counter Chunks (CC), as shown in Fig. 3.4. The procedure is repeated until a single block is generated, i.e., the root. The final nonce value of the root is held on-chip in secure storage. Thus, the tree is composed of DCs (which are the lowest level where the data are stored), and CCs, which are parent nodes that reside on the remaining levels of the tree. The nonce of the final CC is stored on-chip.

VAULT and Morphable Counters The works in [97], [98] suggest means to reduce the depth of an integrity tree by using a different counter configuration and increasing arity of the tree. [97] design a tree with 128-arity, and hence, they can considerably reduce the depth of integrity trees for securing large memories. For protecting a 16GB memory, they were able to reduce the tree size to 4 levels. However, the use of larger arity to obtain shallow integrity trees introduces additional problems such as frequent counter overflowing, which increases the overhead of the scheme. The success of such methods also relies heavily on the counter nodes being cached efficiently, as this contributes to reducing the number of counter writes causing overflows. However, caching counter nodes is not always possible, especially for systems with tight constraints.

ASSURE ASSURE [99] employs smart MAC's and multi-node trees to provide security. Multi-node trees also help in reducing the verification steps by creating multiple roots closer to the leaf nodes that are more frequently accessed. They divide the memory into two groups, HOT and COLD MBG's, and monitor the number of accesses for each group. The root for the HOT MBG is different from the actual root of the tree and is closer to the last level of the tree. Accesses made outside the HOT MBG (i.e., from the COLD MBG) go through the same number of levels as a balanced tree. HOT MBG can be dynamically varied depending on memory access patterns. The success of ASSURE architecture depends on frequent access made to a HOT memory block group. If the majority of accesses are sparsely distributed and do not belong to the HOT memory block, the benefits are limited.

2.3 Discussion

2.3.1 Limitations of Existing Memory Authentication Schemes

Almost all techniques used for memory protection include some form of encryption and an integrity tree as part of the basic framework. Memory encryption deals with encrypting

the contents of the main storage with a cryptographic key that is securely stored on-chip. The common assumption is that anything stored on-chip cannot be tampered. For a fully balanced integrity tree, the number of checks to verify the integrity of each memory block during read accesses and the number of tree updates for each memory block during write accesses correspond to the number of tree levels, which is $\log_2 n$ (where n is the data size) for a fully balanced integrity tree. The additional memory accesses required for runtime verification for a fully balanced integrity tree, therefore, result in high computational overhead. This contributes to excessive performance overhead when n is large. The performance constraints imposed on embedded devices therefore, present a significant challenge for adopting such robust memory security schemes.

2.3.2 Opportunities for Reducing Authentication Overhead

The integrity trees adopted in nearly all the existing works (with the exception of ASSURE [99]) are based on a fully balanced tree (as shown in Fig. 2.2), where the number of verification steps of each memory block are equal. It is a well-known observation that applications running on embedded systems spend 90% of its time on 10% of the code [100]. Hence, not all the memory locations will be accessed equally in typical applications. This can be leveraged upon to generate application-specific skewed integrity trees to reduce the authentication overhead. Also, the sensitivity of the information stored in memories varies, serving as another motivation for designing application-specific memory-protection schemes [87].

The aim of this research is to devise compile-time and runtime algorithms to skew memory integrity trees based on memory access patterns. An example of a skewed memory integrity tree is shown in Fig. 2.3, where we assume that $Data_1$ and $Data_2$ are accessed more frequently. In a skewed memory integrity tree, memory addresses that are frequently accessed will be placed at higher levels of the integrity tree. This reduces the number of

levels required to reach the root node $CC1$ during runtime verification of these frequently accesses memory locations. As such, we expect the overall computational overhead of runtime verification to be reduced compared to its balanced tree counterpart in Fig. 2.2.

Figure 2.2: Fully balanced memory integrity tree

Figure 2.3: Skewed memory integrity tree

To the best of our knowledge, schemes employing a skewed memory integrity tree based on memory access patterns have not yet been studied in detail. The performance impact of skewed memory integrity trees in the presence of caches, multi-core architectures, or their vulnerabilities to attacks such as side-channel and RowHammering attacks present interesting area of studies that have not yet been explored. In addition, such schemes will incur additional overhead to dynamically maintain the integrity tree structure and perform specialized tree traversal. A thorough investigation will be required to understand the

potential and limits of employing skewed memory integrity trees for memory authentication. In the next sub-section, we list a few challenges that must be addressed when customizing memory integrity trees.

2.3.2.1 Research Challenges

The following lists the main challenges for employing customized memory integrity trees:

- **Designing New Tree Structures:** Unlike existing balanced memory integrity trees, a customized memory integrity tree is expected to exhibit an irregular tree structure as the tree can be arbitrarily skewed. In the extreme situation, each leaf node will have different path traversal distance to the root. To create a skewed tree structure that can lead to reduced memory authentication overhead, an understanding of the physical memory regions and the micro-architecture (e.g. cache configurations) is necessary such that efficient grouping of memory blocks into nodes and the optimal placement of the nodes in the tree can be undertaken.
- **Tree Traversal:** For a conventional balanced tree, the nodes are stored starting from the top of the tree down to the leaves in a specific order (as can be seen in Fig. 2.2). Thus, it is easy to determine the position of the parent node based on the child node number. However, with a skewed tree, the position of the nodes will change during runtime and we cannot rely on simple calculations to determine the position of the parent nodes. New tree traversal algorithms and node structures must be developed to ensure that the verification steps from the leaf nodes to the root can be accomplished efficiently.
- **Dynamic Skewing based on Memory Patterns:** The memory access patterns of applications may not be known at compile-time. As such, a runtime system is required to monitor the memory access patterns and make effective tree restructuring

decisions at runtime. These decisions should take into consideration the trade-offs between placing the frequently accessed memory blocks closer to the root (e.g. by increasing the arity of the tree), and the overhead of maintaining the dynamic integrity tree (e.g. ensuring that the overflowing of counters used for encryption is minimized).

- **Complementing Existing Schemes:** Several optimizations on integrity trees (e.g., Bonsai Merkle tree [101], utilizing Tree-Cache [102] and variable arity [98]) have been proposed previously to improve the performance of memory authentication. The proposed dynamic skewed memory integrity tree should be able to complement these techniques to further reduce the authentication time.

2.4 Summary

This chapter provides a comprehensive review of the prevalent methods for addressing the security concerns in embedded systems and memories. The functional security requirements of any embedded system are confidentiality, integrity, and privacy. While some aspects of security have been addressed in the context of traditional general-purpose computing systems, embedded systems usher in many new challenges. This survey highlighted the security-related problems faced by designers of embedded systems in terms of cost, performance, and power. In addition, current trusted computing platforms cannot extend fully beyond the processor core to guarantee secure data storage in external third party memories [26] without incurring high performance overhead. As such, the security and integrity of memory in embedded systems have been a long-standing issue in trusted system design [27].

Current memory-protection schemes rely heavily on a balanced integrity tree [33,98,99] for memory authentication. While attempts have been made to enhance the performance of

runtime memory integrity verification, these methods still suffer from significant overheads due to the need to traverse all levels of the tree for each memory access. Customizing the memory integrity tree based on the memory access patterns provides a promising solution to overcome the large performance overheads. However, there is no existing work that has been undertaken to fully explore the potential and limitation of this approach. In this thesis, we will provide a rigorous investigation on the use of customized memory integrity tree for reducing the runtime overhead. The following chapter describes a novel approach to construct a skewed memory integrity tree for memory protection at compile-time based on the static analysis of memory access patterns.

Chapter 3

Static Skewed Memory Integrity Tree

In this chapter, we propose a framework to construct static skewed integrity trees based on the memory access patterns of an application. As discussed in the previous chapter, applications running on embedded systems spend 90% of their time on 10% of the code [100]. As such, many embedded applications have deterministic execution patterns that can be leveraged upon to generate application-specific skewed integrity trees. To the best of our knowledge, no effort has been undertaken to construct skewed memory integrity trees in a systematic and application aware manner. The main contribution of this chapter is a framework that can automatically generate an application-specific skewed memory integrity tree based on the application characteristics. The proposed framework relies on an offline process to analyze the frequency of memory accesses. The package-merge algorithm is employed to generate a skewed memory integrity tree where the most frequently accessed data are placed at higher levels of the integrity tree. This reduces the verification steps required for the frequently accessed memory locations, which lead to reduced memory authentication overhead.

The work described in this chapter has been published in [1].

3.1 System Overview

3.1.1 Threat Model

The main focus of this work is to prevent bus attacks. Similar to previous works, our threat model assumes that data stored on chip is secure and is resistant to all kinds of attacks. We are concerned with attacks that tamper the memory and/or processor bus, which will enable the attacker to observe and inject manipulated data. Side-channel attacks are beyond the scope of this work. The attacker can choose to tamper the data with any of the following active attack mechanisms:

- **Spoofing:** Attacker exchanges a memory block with a tampered one.
- **Splicing:** Attacker replaces the memory block at address A with a memory block at address B where $A \neq B$.
- **Replay:** Attacker records data at an address and inserts it at the same address at a later point in time. Thus the present value of the data is replaced by an older value.

3.1.2 Security Model

The overview of the security model that is based on the NIOS II processor [103] environment is shown in Fig. 4.1. The integrity tree that has been customized based on the offline profiling of memory accessed patterns is stored in the external memory. The on-chip memory stores the key and root value of the tree. At runtime, the authentication steps after a *Read/Write* request is sent to the Protected SDRAM data region can be described as follows.

Step 1: SDRAM controller forwards the leaf node corresponding to the data address requested to the NIOS II processor for decryption.

Figure 3.1: Security design overview

Step 2: NIOS II sends a request to the Integrity Tree region through the controller to obtain the parent of the node that was just decrypted.

Step 3: The parent node is sent to AES for decryption similar to Step 1.

Step 4: Once both the child and parent nodes are decrypted, they are verified by the Integrity Checker. Steps 2, 3, and 4 are repeated till the root of the tree is reached. If verification fails in any one of these steps, an alarm is triggered.

Step 5: The root of the tree is matched with the Root Node stored on-chip. In case of mismatch, an alarm is raised.

Step 6: If verification is successful, the data requested in Step 1 is sent to the processor pipeline of NIOS II.

A similar process of verification is followed before a *Write* request is implemented by the processor. Any changes to the memory are made after performing an integrity check.

3.2 TEC-tree

For the evaluation of our proposed design, we have chosen the TEC-tree [33] as our baseline. An advantage of the TEC-tree implementation is that the encryption scheme provides confidentiality for all to and from memory traffic by encrypting it before sending to external memory. A TEC-tree is a fully parallelizable tree that consists of both read and write operations. An example TEC-tree is shown in Fig. 3.2

In our analysis, we consider only binary TEC-trees with Data Chunks (DC) as leaves and Counter Chunks (CC) as intermediate nodes, each of b -bit blocks. Each DC is composed of a k -bit data payload and n -bit nonce while each CC is composed of two r -bit counter values and an n -bit nonce as can be seen in Fig. 3.3 and Fig. 3.4. CTR1 and CTR2 refer to the count values of a given CC's child node.

Figure 3.2: A TEC-tree for 8 Data Chunk blocks and 4 levels

N is a nonce which is a "Number used ONCE". The uniqueness of the nonce is ensured by making it equal to the address of the block concatenated with an r -bit counter.

Figure 3.3: A sample Data Chunk

Figure 3.4: A sample Counter Chunk

Initially, the counter is set to zero, and it is incremented every time the processor writes to the block in memory. These counters are local to the particular block. In a TEC-tree, the plaintext is composed of $(D||N$, where $||$ is a concatenation operation), where D is the payload data of k -bits and N is an n -bit nonce. Let us denote the encryption and decryption secret keys as ENK and DEK . After encrypting the plaintext using the key, $C = ENK(P)$, it is near impossible to distinguish the D and N in the ciphertext.

If C' is obtained by flipping exactly one bit in C , there is a high probability that the last n -bits of the plaintext $P' = DEK(C')$ are different from N . This probability depends on the nonce size n . The number of possible plaintext blocks with the same N resulting from the decryption of a tampered C is equal to $2b - n$. Hence, the probability that N remains the same after decryption is $\frac{1}{2n} = \frac{2b - n}{2b}$. This implies that with a sufficiently high value of n , the block is made resistant to tampering attacks.

Authentication Reading and writing to the memory is performed using the two functions: *ReadNCheck* and *WriteNUpdate*. These functions perform authentication while reading and updating the tree accordingly on load/store requests.

ReadNCheck: On a read request, the DC requested by the user, is loaded from memory and decrypted. The corresponding parent CC is also fetched and decrypted to obtain the

corresponding counter value. If both values match, data is passed to the processor. The authentication will continue recursively until the root. Should there be any mismatch in the counter values, a warning is sent to the processor, and it terminates the execution.

WriteNUpdate: For a write request, DC to be updated is first authenticated by performing the *ReadNCheck* function, and then updated by overwriting the data. The counter value is incremented by one followed by recursively updating the parent CC counters until the root. Finally, the nonce stored on-chip is incremented by one. For a L -level tree, $L+1$ comparisons are required for a single authentication. The last comparison is for checking the on-chip value.

It is worth mentioning that the proposed method for customizing memory integrity trees can be applied to all existing memory integrity trees. The objective of this work is to develop an algorithm to alter the structure of the tree by giving weight to each of the leaf nodes. The tree will be modified such that nodes with a higher weight have a shorter path and thus can reach the node in a shorter period. This will eventually lead to a decrease in the runtime of applications that employ integrity trees.

3.3 Proposed Method

3.3.1 Static Skewed Memory Integrity Tree Model

In this section, we will introduce the modifications made to the TEC-tree [33] to enable the construction and deployment of a customized skewed memory integrity tree. Note that unlike a binary tree structure, a skewed tree implementation as shown in Fig. 2.3 increases the complexity of tree traversal since the determination of the location of the parent node (or their memory address) for child node i cannot be easily achieved with $\lfloor \frac{i}{2} \rfloor$. To simplify this problem, the proposed approach maintains a balanced physical tree form in memory. But it allows for empty nodes between the leaf nodes and their respective parents to emulate a skewed tree structure. Hence, in the modified TEC-tree

Figure 3.5: Example of a modified TEC-tree

implementation, the parent node may not necessarily be connected directly to the child nodes in the tree, even though we are effectively utilizing a balanced tree structure. DC's are placed on the last level of tree as the leaf nodes. While performing verifications, the parent node may be accessed by skipping some levels in the tree. This requires a different method for calculating parent node (address) of the leaf nodes. Primarily, the approach relaxes the restriction imposed on fully balanced trees that necessitates tree traversal to be performed at all levels of the tree for each memory block.

Numbering the nodes in the same manner as a balanced tree simplifies the calculation of the parent address for any given node. Except for the leaf nodes, the parent of any node i can still be calculated as $\lfloor \frac{i}{2} \rfloor$. The parent node numbers of the leaf nodes are directly stored in the DC (as shown in Fig. 3.5). This enables the parent nodes, that are separated from their children by multiple levels, to be accessed directly. An illustration of the modified TEC-tree with 8 leaf nodes is shown in Fig. 3.5. Parents of the 8 leaf

nodes are located at different tree levels. The runtime verification of Node 16 and Node 17 requires only two accesses of CCs (i.e., Node 2 and Node 1). The dotted line represents the first jump that nodes take to reach their respective parents. Compared to a fully balanced tree implementation, the modified TEC-tree skips two levels during verification of Node 16 and Node 17.

To implement the modified TEC-tree, additional information is required in the DCs as shown in Fig. 3.5 (the format of the CCs in Fig. 3.5 is the same as [33]). In particular, two extra fields, i.e. *Parent* and *LR*, have been added in the DC. The *Parent* field specifies the parent node while *LR* indicates the child position, i.e., which node is left or right child of its parent. The *Data* field holds the actual data. The *Node* and *Count* fields are concatenated to form the nonce. *Node* is the node number of the tree, and *Count* represents the number of write operations on that particular node. As the node number is specific to each node of the tree, the nonce for each node is unique. The CC has two fields *Count1*, and *Count2*, which store the *Count* values of its right and left child respectively. The nonce of the CC is formed in the same manner as a DC. For example, Node 16 has 2 stored in its *Parent* field meaning that node 2 is its parent. Node 16 being the left child of node 2 has its *LR* bit as 0 and similarly for node 17, also the child of node 2, the *LR* bit is 1. The *Count* field at node 16, i.e., 5 is the *Count1* placed at node 2 and similarly the *Count* of node 17, i.e., 4 is set at the *Count2* field of node 2. The *Count* value of node 2 represents the writes made on all of its children and hence, is set to 9.

Memory Overhead The memory access overhead due to the CC's and DC's has been discussed in [33], and is equal to:

$$m = \frac{l + ((d + count) * Arity)}{(l * (Arity - 1))} \quad (\text{Eq. 3.1})$$

where l is the size of data, and d is the number of bits used to represent that address as can be seen in [33]. The additional storage required for a skewed tree over a balanced tree is:

$$m' = (\log_2(n) + 1) * d \quad (\text{Eq. 3.2})$$

n being the number of data elements. As the table is being stored in external memory, this additional storage requirement does not pose a serious concern.

3.3.2 Static Skewing Algorithm

The framework consists of three parts: (i) offline profiling for weight calculation, (ii) tree formation, and (iii) authentication. It is noteworthy that the method can be easily extended for code integrity protection.

3.3.2.1 Offline Profiling

Application profiling is performed by running the application to extract all the data memory addresses whenever a read or write operation is performed on the data. This process calculates the frequency of accesses for each memory address. Based on this information we can calculate the weights of each data using (Eq. 3.3) to (Eq. 3.5).

Weight Calculation The weight of each node is calculated based on the amount of time it requires to perform a read and write operation. For a read, verification is performed at each level. Thus, the time required depends on L , the number of levels of the tree as all data chunks are invariably placed on the last level as leaf nodes. Furthermore, at each level, *Count* value of both the node and its parent is required to perform verification; this requires decryption to be performed for both the nodes. Both these sum up to form the weight required for a *ReadNCheck* function as shown in (Eq. 3.4). *WriteNUpdate* can be divided into two parts: 1) *ReadNCheck*, and 2) Updating. A write operation on any

memory location is performed only after it has been ensured that the location has not been tampered with before. This requires an additional *ReadNCheck* step. Updating can further be broken down into two parts: 1) Updating the DC, and 2) Updating the CC. Updating the DC requires overwriting the *data*, and *Count* of the node. Thus, first the node is decrypted to extract the required fields followed by updating them, and finally encrypted again. Updating the CC has to be performed at all the remaining levels, i.e., $L - 1$. To update the *Count* value of a CC, we also need to extract the updated *Count* value from its child, to make sure they match. Hence, an extra decryption is performed for this step at each level. The time spent at each level during a read or write can be calculated using the formulae given below. The weight is calculated based on the time spent for a balanced tree. L - number of levels of the balanced tree; t_{up_DC} - time to update a Data chunk; t_{up_CC} - time to update the Counter Chunk; t_d - constant time needed for decryption using AES algorithm; t_e - constant time needed for encryption using AES algorithm.

$$Weight = ReadNCheck + WriteNUpdate \quad (Eq. 3.3)$$

$$ReadNCheck = (L) * (2 * t_d) \quad (Eq. 3.4)$$

$$WriteNUpdate = ReadNCheck + t_{up_DC} + t_{up_CC} \quad (Eq. 3.5)$$

$$t_{up_DC} = t_d + t_e \quad (Eq. 3.6)$$

$$t_{up_CC} = (L - 1) * (2 * t_d + t_e) \quad (Eq. 3.7)$$

These weights will decide the node placement for the tree. We will run the package-merge algorithm as described in the next section on these weights. The underlying principal is that the node with a higher weight should have a shorter path to the root of the tree.

3.3.2.2 Package-Merge Algorithm - Tree Formation

The package-merge algorithm [104] is applied to construct the skewed memory tree. The nodes are placed in memory before running the main application code. The time complexity of this algorithm is $O(nL)$, where n is the number of nodes and L is the number of levels of the tree. The weights are first sorted in an increasing order, and then the problem can be solved in linear time. The pseudo-code used to perform the calculations is shown in Algorithm 3.1.

The inputs to the problem are a set I , a number, X , and the optimal height of the tree.

A node-set I is described as a set of ordered pairs (i, l) where $1 \leq i \leq n$ and $1 \leq l \leq L$. i being the node number and l representing the level of the node on the tree. We will use the notation l_i to denote the level of node i on the tree. Each element of I has a width and weight, where width is defined as 2^{-l} , and weight, w_i , is a non-negative number which in our case describes the frequency of data access. The elements in the set are sorted in an increasing order by weight.

X is a non-negative number denoting the *totalwidth* i.e. $\sum_{i=1}^n 2^{-l_i}$, where l_i is the level of node i . Initially as l_i is zero for all the nodes, $X = n$. Whenever the level of a node increase by 1, X will be reduced by 2^{-1} . The diadic expansion of X is its representation as a power of 2. For example, the smallest item in the diadic expression of 12 is 4 since $12 = 8 + 4$. We should note here that $X \leq 1$ will give us a prefix free code. Given a prefix free code, creating a binary tree is straight-forward.

Algorithm 3.1: Package-Merge (I, X)

```

begin
   $S \leftarrow \phi$ 
  for all  $d, L_d \leftarrow$  list of items of width  $2^{-d}$ , sorted by weight
  while ( $X \geq 1$ ) do
     $minwidth =$  smallest item in the dyadic expression of  $X$ 
    if  $I = \phi$  then
      | return "No Solution"
    end
    else
       $d \leftarrow$  the minimum such that  $L_d$  is not empty
       $r \leftarrow 2^{-d}$ 
      if ( $r \geq minwidth$ ) then
        | return "No Solution"
      end
      else
        if  $r = minwidth$  then
          | Delete the minimum weight item from  $L_d$  and insert it into  $S$ 
          |  $X \leftarrow X - minwidth$ 
        end
      end
       $P_{d+1} \leftarrow \text{PACKAGE}(L_d)$ 
      discard  $L_d$ 
       $L_{d+1} \leftarrow \text{MERGE}(P_{d+1}, L_{d+1})$ 
    end
  end
  return  $S$  is the optimal Solution
end

```

Another input that is needed by the package-merge algorithm is the optimal height of our tree. Skewing the tree will, in most cases, end up increasing the height. For every level that is increased, some paths get shorter, and some get longer. Thus, skewing a tree beyond a certain height can degrade performance. To calculate the most optimal height, the theorem stated in [105] is applied, i.e., the upper bound for the height of an optimal weighted path length is $2 + H$ for a binary tree. H is the entropy of the frequency distribution of the data nodes. The maximum value of variable d in Algorithm 3.1 can attain is $2 + H$. Running the design on the upper bound is a safe option and gives us a fair estimate to evaluate the design. It is worth mentioning that this theorem is only applicable to binary trees, i.e., trees with arity 2. If we extend this work on trees with higher arity, we will need to adopt some other heuristic measure to find the most optimal height.

The objective of this problem is to finally attain a set S of minimum total weight (for a tree it will be the sum of weighted path length of the nodes, $\sum_{i=1}^n w_i * l_i$) whose width sums to X . The optimal solution will have a *totalwidth* equal to 1, as this would give us a prefix-free code.

3.3.2.3 Authentication

The following steps are performed for reading and writing to the memory blocks of the modified TEC-tree. On similar lines as for the baseline model discussed above for a read, the *ReadNCheck* function is performed and for a write, the *WriteNUpdate* function is performed.

ReadNCheck: When a DC is read from memory, it is first decrypted, and the corresponding *Parent* is extracted. Based on this, the parent CC is fetched and decrypted to obtain the corresponding counter value *Count*. If the *Count* of the CC and DC matches, the authentication will be repeated at the higher level. From this point onward, the

parent node of i can be determined using $\lfloor \frac{i}{2} \rfloor$. The verification is performed up till the root if there is no mismatch in the *Count* values at any level. In case of a mismatch, the read access is terminated. If the *Count* at the root node matches with the value stored on-chip, the authentication is successful.

WriteNUpdate: During a memory write operation, the DC to be updated is first authenticated with the help of the *ReadNCheck* function described above, and then updated by overwriting the data. For updating the block, it is first decrypted, and the data part of DC is extracted and replaced with the new data. The *Count* value in the DC is then incremented by one. Updating of *Count* values is recursively performed from the parent CC up till the root.

3.4 Implementation Example

We design a tree for the following data consisting of 8 data elements (i, j) where i is the data and j is the weight associated with its memory location. The weights have been calculated using [Eq. 3.3](#) to [Eq. 3.5](#) as described in [Section 3.3.2.1](#) as:

$$\{(3,1);(0,1);(1,2);(1,2);(1,3);(1,3);(7,4);(10,4)\}$$

The height of the tree is 4. The package-merge algorithm is performed on the weights $\{1,1,2,2,3,3,4,4\}$ as illustrated in [Fig. 3.6](#).

Figure 3.6: Package-Merge Algorithm

The step PACKAGE is performed to form a list P_{l+1} from L_l by combining items in consecutive pairs, starting from the lightest. Fig. 3.6 explains this step visually. The dotted pairs are the packages that are formed. Thus the package P_2 is formed from list L_1 on level 1 by combining the values of the dotted pairs. Combining is done by simply adding up the weights of the elements of each pair. Thus P_2 is the sum of the elements of all the dotted pairs on L_1 i.e. $\{2,4,6,8\}$. The MERGE step is the usual merging of two sorted lists. The successive lists in Fig. 3.6 are formed by performing the MERGE step. In general, the list L_l is created by forming a package list, P_l , from list L_{l-1} and merging it with a copy of the first list developed. As shown in Fig. 3.6, L_2 is formed by merging P_2 (i.e. $\{2,4,6,8\}$) and the original elements i.e. $\{1,1,2,2,3,3,4,4\}$. The dotted line in Fig. 3.6 points to each package and the arrow points to its sum. The number of times the PACKAGE and MERGE steps are performed depends on the height, L , of the tree.

We define a term *active* leaves as the first $2 * n - 2$ items on the last list. The rationale behind this is to make sure we get $X \leq 1$. The solution is stored in the list $S = [s_i] \ i \in 1 \dots n$ and $1 \leq s_i \leq L$. s_i is set to the number of active leaves corresponding to each original

element. The active leaves for our example are the first 14 elements on list 5, as shown in Fig. 3.6. They have been shaded and corresponding to each of them, their s_i has been counted. Counting backward, the sum count of the number of times the original items are packaged to get items on the final list is calculated. As an example, consider the first two elements, $\{1,1\}$ on the final list. They are nothing but the original items. Hence, in Fig. 3.7, the respective original item is ticked once. Moving on, the element on the list $\{2\}$ is a package of the original $\{1,1\}$. Thus, the corresponding original items are ticked again. Each tick will essentially represent the item moving one level below. Similarly, the entire final list of 10 elements is traversed in the same manner and the final level are shown in Fig. 3.7.

	Weights							
	1	1	2	2	3	3	4	4
	✓	✓	✓	✓	✓	✓	✓	✓
	✓	✓	✓	✓	✓	✓	✓	✓
	✓	✓	✓	✓	✓	✓		
	✓	✓	✓	✓				
5	4	4	4	4	3	3	2	2

Figure 3.7: Calculating levels

Thus, S for the above example is $\{4,4,4,4,3,3,2,2\}$ which implies node i is to be placed on level s_i .

The final solution, as depicted in Fig. 3.5, is obtained using this calculation. The level each item should be placed for a 4-level 2-ary tree is decided. Issues with weight $\{1,1,2,2\}$ (i.e., Node 20 - Node 23) should be placed at level 4, and their parents are thus placed at level 3. Items with weight $\{3,3\}$ (i.e., Node 18 and Node 19) should be placed at level 3, and therefore the nodes on level 2 are their parents. Items with the maximum weight (i.e., Node 16 and Node 17) should be at level 2 (closest to the root), and hence their parent are placed on level 1, making the highest jump of two levels.

3.5 Performance Evaluation

We evaluated the performance benefits of the proposed method using the Altera DE2 board and the Qsys tool in Altera Quartus II, v12.1. The system consists of a NIOS II Processor operating at 50MHz, On-Chip Memory, JTAG-UART for connection between the system and the board, SDRAM Controller for using the SDRAM, Clock Series for DE-series Board Peripherals, and Performance Counter Unit for measuring the performance statistics such as time elapsed and number of clock cycles.

Table 3.1: Time taken to create static skewed tree

No of Data Nodes	4	8	16	32	64
Time (Sec)	0.420	0.702	0.907	1.32	1.94

For our experiments, we ran six deterministic applications from the CH-Stone and SNU Real-Time test benches [106, 107]. The applications are tested: 1) using a balanced TEC-tree, and 2) using the modified TEC-tree as described in Section 3.3.1. We measured the time required to run the benchmarks. The tests were done using two different data sets to get a fair estimate of performance. Data set 1 was part of the original test-benches. Data Set 2 is synthetically generated using random values and have twice the size of data set 1. Each data set was initially profiled offline for weight calculations. The results are shown in Fig. 3.8. The static skewed integrity tree achieves a significant performance improvement of 18% over the conventional balanced tree. The data set used vary in size from 8 to 32 elements. Our goal was to isolate the effect of having a skewed tree in an application, and thus the performance gain is calculated only for the application runtime and does not consider the time used to create the skewed tree. Tree creation depends on the number of nodes which are varied from 4 to 64 as is shown in Table 3.1. The platform used for both tree creation and running the application is the same as discussed earlier.

Note that the amount of gain will depend directly on the kind of application and its memory usage pattern. Thus, we expect to see variation for different applications. Applications dealing with data sets with more substantial variance in the memory access frequencies are expected to result in more significant performance benefits.

Figure 3.8: Performance gain in comparison to a balanced tree

3.6 Summary

We presented a compile-time method for generating a customized skewed memory integrity tree based on the frequency of memory accesses of the application. We modified the TEC-tree to allow for the more frequently accessed memory blocks to traverse a shorter verification path. This is achieved by including additional fields in the data chunk to indicate the parent of the leaf nodes. We formulated the weights to reflect the memory access overheads of read and write operations for each memory block. The package-merge algorithm was adapted to place the weighted memory blocks in the skewed tree so that memory blocks with larger weights will require less verification steps. The proposed

method can be extended for other types of memory integrity trees. Experimental results based on widely used benchmarks demonstrate the effectiveness of our approach. The security analysis of the skewed memory integrity tree will be discussed in Chapter 6.

The methods described in this chapter are limited to applications wherein the memory access patterns are known at compile-time. In the following chapter, we propose a dynamic skewed memory integrity tree that adapts to the runtime memory access patterns, and discuss its performance benefits over the static approach.

Chapter 4

Dynamic Skewed Memory Integrity Tree

In the previous chapter, we introduced a static skewed memory integrity tree where the tree is constructed offline based on the frequency of memory accesses derived from static application profiling. Tree nodes that are accessed frequently are placed closer to the root based on the static application profiles. However, the tree structure remains unchanged at runtime. While the work in the previous chapter demonstrated the potential of customizing the memory integrity trees based on memory access patterns, the static approach is only applicable to scenarios where the memory access patterns of the applications are known beforehand.

In this chapter, we propose a method that dynamically restructures the memory integrity tree based on the runtime memory access patterns. In particular, our approach places data elements with the same memory access frequency together in a single node/set, leading to efficient tree storage. As the memory access patterns change, nodes in the tree are repositioned based on their frequency of access. Nodes with higher frequency are placed closer to the root. Unlike the method presented in the previous chapter, the proposed dynamic skewed memory integrity tree method does not assume that the

The work described in this chapter has been published in [2].

Figure 4.1: System design overview

memory access patterns are known a-priori. We demonstrate the effectiveness of the proposed approach for runtime verification on the Altera NIOS II processor with an external DRAM.

4.1 System Overview

The overview of the system that is based on the NIOS II processor environment is shown in Fig. 4.1. The integrity tree is stored in the external memory. The on-chip memory stores the key and root value of the tree. The NIOS II processor will perform the memory data authentication process which consists of: (i) encryption, (ii) integrity check and (iii) skewing during runtime. The tree operations after a *Read/Write* request is sent to the Protected Data region can be described as follows:

- Step 1: SDRAM controller forwards the leaf node corresponding to the data address requested to the NIOS II processor for decryption.
- Step 2: NIOS II sends a request to the Integrity Tree through the controller to obtain the parent of the node that was just decrypted.

- Step 3: The parent node is sent to AES for decryption similar to Step 1.
- Step 4: Once both the child and parent nodes are decrypted, they are verified by the Integrity Checker. Steps 2, 3, and 4 are repeated till the root of the tree is reached. If verification fails in any one of these steps, an alarm is triggered.
- Step 5: The root of the tree is matched with the Root Node stored on-chip. In case of mismatch, an alarm is raised.
- Step 6: If verification is successful, the data requested in Step 1 is sent to the processor pipeline of NIOS II.
- Step 7: *Update*: If the request in Step 1 is a write request, NIOS II initiates an Update command through the SDRAM controller to restructure the Integrity Tree.

The threat model remains the same as explained in the previous chapter (Section 3.1.1).

4.2 Proposed Method

4.2.1 Dynamic Skewed Memory Integrity Tree Model

The proposed tree structure uses nodes as *sets* that store a group of symbols rather than an individual symbol. Each data chunk (or set) will store all the locations that have the same frequency of usage. Therefore, the number of tree data chunks is equal to the memory access frequencies encountered so far instead of the number of data elements as in the static skewed tree structure described in the previous chapter.

The data elements are placed as leaf nodes. Fig. 4.2 shows how the elements have been grouped based on their frequency of usage. The format of the data and counter chunks has been slightly modified from Chapter 3. Let n_i denote the i^{th} node of the tree. New slots of parent, sibling, LR have been added to the original data chunk. Thus $n_i.p$, $n_i.s$, $n_i.LR$, $n_i.f$ denote the parent, sibling, side, frequency of the node, respectively.

Figure 4.2: Example of a dynamic tree structure

$n_i.p$ holds the node number of the parent and $n_i.s$ holds the node number of the sibling. LR denotes whether the node is a left child or a right child to its parent. $n_i.f$ denotes the frequency of access for the elements stored in the node-set. $n_i.Symbol$ denotes the number of data elements in the node-set. The weight of the node is denoted by $n_i.w = n_i.Symbol * n_i.f$. As can be seen in Fig. 4.2 the concatenation (Node||Weight) in the counter and data chunks serves as the nonce, which helps protect against replay attacks. Fig. 4.2 shows an example tree structure for data elements: (0,1,1,5,6,8,9). The attributes of n_4 are: $n_4.p = 3$, $n_4.s = 5$, $n_i.LR = 0$, $n_i.f = 2$, $n_i.Symbol = 2$. Thus, weight $n_i.w = 4$.

The counter chunk has the same format as the data chunk nodes. As the tree is not a fully balanced tree where the location of the parent of a child node n_i can be easily computed by $\lfloor \frac{i}{2} \rfloor$, we need to store parent and sibling address of each node. The counter chunk stores the weight of its two children instead of data elements. Its weight is the sum of the weight of its children. In Fig. 4.2, n_3 is a counter chunk. It stores the weight of its

children (i.e. n_4 and n_5) and its own weight, $n_3.w$, is 7 (i.e $3 + 4$). The same goes for n_1 , which is also a counter chunk.

We should note here the data chunk (n_5) with $n_5.f = 3$ is placed at level 2 as compared to data chunk (n_2) with $n_2.f = 1$ placed at level 1. This is because $n_2.w$ is greater than or equal to $n_4.w$ and $n_5.w$.

The proposed tree structure incurs a memory overhead in the form of a lookup table (LUT). Unlike the fully balanced tree structure, where each element was stored in the same order as it occurs in the application following a natural order, this tree stores elements in a different fashion giving priority to frequency. The LUT would be storing the node number along with an element number (to help determine which element it resides in the particular set). If the number of nodes of the tree is n , the memory overhead can be calculated as: $3 * \log_2 n$. It is worth noting that the memory savings using this tree structure compensates for the overhead. For this example, the number of nodes for a balanced tree would have been 15 compared to the 5 in our case. The LUT for data elements in Fig. 4.2 is shown in Table 4.1.

Table 4.1: Lookup table

Index	Node No.	Element No.
1 $\{0\}$	4	1
2 $\{1\}$	4	2
3 $\{1\}$	2	1
4 $\{5\}$	2	2
5 $\{6\}$	5	1
6 $\{8\}$	2	4
7 $\{9\}$	2	3

4.2.2 Dynamic Skewing Algorithm

The proposed method for dynamically skewing the memory integrity tree is shown in Algorithm 4.1. The tree formation begins with a single node comprising of all the data

Algorithm 4.1: Dynamic Skewed Tree

```
begin
  Initialize tree with single node set having all data elements
  while System is working do
    if read_request(addr) then
      n ← LUT(addr)
      ReadNCheck(n)
      Set_Migration(n)
    end
    if write_request(addr) then
      n ← LUT(addr)
      WriteNUpdate(n)
      Set_Migration(n)
    end
    Rebalance_flag ← rebalance_check(n);
    if Rebalance_flag then
      ShiftUp(n);
    end
  end
end
```

elements with frequency 0. When any data element is accessed, its frequency is increased by 1, which triggers the process of tree update. An alternative initialization method will be to start with the static skewed memory integrity constructed using the method described in the previous chapter. During runtime, whenever the system gets a read memory request, the processor matches the address with the LUT to obtain the node number of the tree. This is followed by verifying all the nonces on the path up to the root. If verified, the data is passed to the processor and that element is migrated to a new node-set. A similar procedure is followed for a write request. After the access, the tree is checked for rebalancing. If a particular node with large weight has been shifted to the lower levels of the tree, the tree is rebalanced by relocating the nodes accordingly.

The criterion that has been used for rebalancing as described in [108] is as follows:

$$\text{if } (n_i.w > ((n_i.s).w + 1) \wedge (n_i.w > ((n_i.p).s).w)) \text{ then rebalance}$$

The above criterion states that if the weight of a node is greater than the weight of its sibling node by at least two and is higher than the weight of its uncle node, then it should be relocated.

The dynamic tree algorithm performs two other operations to restructure itself after a memory access: Set migration and rebalancing.

- *Set Migration*: Migration happens when a data element moves from one set to another. Essentially, the aim is that elements with the same probability of occurrence remain in the same set. Thus, whenever an element is called (thereby increasing its probability of occurrence), it has to be migrated to a new set with frequency 1 more than its current set. If such a set does not exist, a new set is created with that element being its first member.
- *Rebalancing*: This operation maintains and alters the tree structure. The nodes are moved around whenever a region in a tree goes out of balance. If a certain node has higher weight than its neighboring nodes, then it ought to be moved higher up in the tree.

The *Rebalancing* criterion is checked for all nodes up to the root. Rebalancing broadly consists of shifting up the node closer to the root. It does so in two steps:

- Exchanging subtree with its uncle
- Rotation

The operations have been depicted in Fig.4.3. Every time an element is accessed, its frequency goes up by one, and the weight of the node increases. It is checked against the *Rebalancing* criterion. If the test is satisfied, the *ShiftUp* operation is performed. Please note that A, B, C (as shown in Fig. 4.3) are all sub-trees and not single nodes. Thus, the nodes are exchanged along with their children. The pseudo-codes used for migration and shifting up are described in Algorithm 4.2 and Algorithm 4.3 respectively.

Figure 4.3: Shifting up of sub-tree C: (a) Exchanging with uncle A, (b) Rotation, (c) Final state

The operation of migration for an element has a total time complexity of $O(\log k)$ in the worst case, where k is the cardinal of the set. The update procedure can also be done in logarithmic time as shown in [108].

4.2.2.1 Authentication

The steps performed for reading and writing are the same as the static skewed tree. For a read, *ReadNCheck* is performed and for a write, *WriteNUpdate* is performed.

- *ReadNCheck*: The DC is loaded from memory and decrypted while the corresponding CC is fetched to obtain the corresponding counter value. If the two values match, the same process is applied to the next level. The remaining branch authentication until the root is carried out and in case of any nonce mismatch, a warning is sent to the processor else the data is passed to the processor. Once moved, the data is migrated to a new set with higher frequency. The tree is checked if there is any imbalance and if *Rebalancing* is required.
- *WriteNUpdate*: The DC to be updated is authenticated using the above process following which the DC's payload is updated and concatenated with a new nonce (incremented counter value). This is followed by updating the parent CC and so on

Algorithm 4.2: Set Migration $\{a : \text{data element}\}$

```
begin
  Q, P : pointers to Nodes
  P  $\leftarrow$  find (a)
  Q  $\leftarrow$  find (P's frequency +1)
  if Q  $\neq \emptyset$  then
    remove a from P's set
    P's weight = P's weight - P's frequency
    add a to Q's set
    Q's weight = Q's weight + Q's frequency
    ShiftUp (Q)
    if P =  $\emptyset$  then
      | remove P from the tree
    end
  else
    | ShiftUp(P's Sibling)
  end
end
else
  create a new node T
  T's right child is a new node N
  T's left child is P
  N's set = a
  N's weight = P's frequency +1
  N's frequency = P's frequency +1
  replace the old P in the tree by T
  remove a from P's set
  P's weight = P's weight - P's frequency
  if P =  $\emptyset$  then
    | remove P from the tree
  end
  else
    | ShiftUp(P's Sibling)
  end
  ShiftUp (T)
end
end
```

until the root. All loaded chunks are authenticated before being updated. Once updating is complete, the data is migrated to a new set with higher frequency. The

Algorithm 4.3: ShiftUp $\{T : \text{pointer to node}\}$

```

begin
  while  $T$  is not the root do
     $T$ 's weight =  $T$ 's right child weight +  $T$ 's left child weight
    if  $(T$ 's weight  $> T$ 's sibling weight + 1)  $\wedge$  ( $T$ 's weight  $> T$ 's uncle weight)
    then
       $Q \leftarrow$  parent of parent of  $T$ 
      exchange  $T$  with  $T$ 's uncle
      exchange  $Q$ 's right and left children
      update  $T$ 's ancient parent's weight
    end
     $T \leftarrow T$ 's parent
  end
end
    
```

tree is checked if *Rebalancing* is required.

4.3 Implementation Example

Figure 4.4: Tree migration and rebalancing steps

We consider the example in Fig. 4.2 with the following data symbols: $\{0,1,1,5,6,8,9\}$.

Suppose that the elements are accessed in the following order: $\{0,1,1,5,6,8,9,6,0,1,6,1,6\}$.

Note that all the 7 elements are accessed once in the beginning. NE_i denotes the number of data elements in the node. We begin our tree design with a single set consisting of all the elements and frequency 1 as can be seen in n_2 on Tree 1 in Fig. 4.4. It has a weight of 7, with 7 elements each having occurred once. From the access log we note the next element being called is $\{6\}$. As currently there is no set with frequency 2 we will need to create a new set. This is created in Tree 2. As $\{0\}$ and $\{1\}$ are called, they are migrated from set of frequency 1 to 2. This results in the tree structure shown as Tree 4 which consists of 3 nodes, n_3 with 3 elements and weight 6 and n_2 with 4 elements and weight 4. Next, $\{6\}$ is called again. We will need to create another new node, this time with frequency 3. An additional step needs to be performed i.e. we need to build a new counter chunk and the new node of frequency 3 will be one of its children as can be seen in Tree 5. Next $\{1\}$ is called again. It will now migrate to n_5 . The tree at this point can be seen as Tree 6 in Fig. 4.4. It now needs rebalancing as n_5 with weight 6 satisfies the shift up criterion. After performing the tree shift up steps as shown in Fig. 4.3, Tree 7 is obtained. In the final state, it can be seen that the element $\{6\}$ and $\{1\}$ which are being called the maximum number of times are placed closest to the root to reduce the time required for runtime verification.

4.4 Performance Evaluation

We evaluated the performance benefits of the proposed method using the Altera DE2 board and Qsys tool in Altera Quartus II, v12.1. The system consists of a NIOS II processor operating at 50MHz, On-Chip Memory, JTAG-UART for connection between the system and the board, SDRAM Controller for using the SDRAM, Clock Series for DE-series Board Peripherals, and performance counter unit for measuring the performance statistics such as time elapsed and number of clock cycles.

In our experiments, we ran six applications from the CHStone and SNU Real-Time test benches. The applications are tested on: 1) balanced TEC-tree, 2) static skewed tree, and 3) proposed dynamic skewed tree. Fig. 4.5a shows number of tree levels that are accessed for the various methods considered. It can be observed that the proposed dynamic memory integrity tree results in the least number of levels accessed. On average the number of levels reduced by 35% compared to a balanced tree and 25% compared to the static skewed tree. The performance overhead of the proposed method for constructing and maintaining the dynamic tree is shown in Fig. 4.5b. On average the overhead constitutes to about 30% of the overall runtime. Next, we measured the run time for all the six applications on the different tree structures. In particular, we observe on average an improvement of 30% over a full balanced tree and an average improvement of 20% over the static skewed tree. The results are shown in Fig. 4.5c.

It is clear from the results that dynamically skewing the tree helps to reduce the authentication time. This gain can be directly attributed to the reduction in the number of tree levels accessed while performing read and write operations. The variation in the runtime reduction is due to the nature of the application and their memory access patterns.

4.5 Summary

This chapter describes the proposed algorithm used to construct a dynamic skewed memory integrity tree. The tree is restructured during execution based on memory access patterns. Data nodes that have the same frequency are placed together in a single node on the tree. Nodes are repositioned in the tree at runtime to ensure that those with higher frequency are placed closer to the root. This reduces the time required to perform memory authentication on those nodes. The experiment results show improvement in runtime of 30% compared to a balanced tree approach due to the reduction in the number of tree

levels that are accessed for all read and write operations. The methods for static and dynamic skewed memory integrity tree that have been described so far do not consider the effects of caches in the system. In the next chapter, we will propose a framework that combines the dynamic skewed memory integrity tree with cache-aware optimizations and present the improvement in performance and energy overhead of memory authentication. The security analysis of the skewed memory integrity tree will be discussed in Chapter 6.

(a)

(b)

(c)

Figure 4.5: Performance evaluation: (a) Number of tree levels accessed, (b) Time overhead for dynamic tree, and (c) Authentication time

Chapter 5

Architecture-Aware Dynamic Skewed Memory Integrity Tree

In the previous chapter, we presented our initial approach for dynamically skewing the memory integrity tree at runtime based on the memory access patterns. This is achieved by placing data elements with the same memory access frequency together in a single set, with each set being treated as a single unit for encryption. Each set is associated with a frequency, and data elements migrate among the sets as their access frequency changes. Although this method achieved higher performance gains compared to the static memory integrity tree approach, it suffered from several drawbacks, which restricts the method from practical realization. Firstly, the number of data elements that can be grouped in a set is limited by the size of the encryption blocks. This could lead to scenarios where multiple sets are associated with the same frequency, and the tree-nodes cannot be optimally placed in the tree structure to reduce the authentication steps. This leads to additional complexity during tree traversal and result in lesser performance gains. Secondly, our prior work requires an additional on-chip memory overhead in the form of a lookup table (LUT) to retrieve the tree locations in memory. This storage overhead is considerable. If the number of nodes of the tree is N , the memory overhead of the LUT

The work described in this chapter has been published in [3].

is $(3 * \log_2 N)$ bits. Finally, the designs of our tree structures in our previous work are agnostic to the cache, and the experiments were undertaken with the assumption that the target system does not have a cache. Moreover, only the runtime gains are measured, and the power/energy is not evaluated.

The proposed framework in this chapter overcomes the limitations of our prior work with an entirely different tree structure design and a new method of dynamically skewing the tree at runtime. We introduce a new node structure for the memory integrity tree that contains all the necessary information to perform dynamic restructuring of the tree based on the memory access patterns. The new node structure and the grouping of data elements in continuous memory addresses into a single leaf node, removes the need for a large on-chip LUT and enables the integrity tree to be restructured dynamically with low-performance overhead. Unlike our static tree described in Chapter 3, we do not require any off-line static analysis to be performed on the applications. Besides, the proposed framework customizes the integrity tree node structure based on the cache configuration of the target processor to take advantage of speculative authentication. The number of data elements in a leaf node is determined based on the cache line size. This scheme exploits the data locality of reference, where during a cache line replacement, only a single authentication step is made for multiple data elements that are likely to be accessed soon.

5.1 Overview of Proposed Security Model

The threat model is the same as previously explained in 3.1.1. In Fig. 5.1, we show our security design model. The model consists of a Memory Controller (MC) which is responsible for data authentication. The entire memory can be split into three distinct regions: a section consisting of sensitive data that needs protection (i.e., Protected DATA),

a section of unprotected data, and a region for storing the integrity tree. The protected data and the integrity tree region will be encrypted with AES.

Figure 5.1: Security design overview

The tree operations after a *Read/Write* request is sent to the Protected Data region can be described as follows with the help of Fig. 5.1:

- Step 1: The leaf node corresponding to the data address requested is sent to AES for decryption.
- Step 2: MC sends a request to the Integrity Tree to obtain the parent of the node that was just decrypted.
- Step 3: The parent node is sent to AES for decryption.
- Step 4: Once both the child and parent nodes are decrypted, they are verified by the Integrity Checker. Steps 2, 3, and 4 are repeated till the root of the tree is reached. If verification fails in any one of these steps, an alarm is triggered.
- Step 5: The root of the tree is matched with the Root Node stored on-chip. In case of mismatch, an alarm is raised.

Step 6: If verification is successful, the data requested in Step 1 is sent to the cache. Any further access to this address will be made from the cache itself and thus will not need to be verified again.

Step 7: *Update*: If the request in Step 1 is a write request, an Update command will be sent from the MC to the Integrity Tree region to restructure the tree.

As discussed in earlier chapters, an integrity tree will be formed with hashes of the protected data region as the leaf nodes. Whenever a memory request is made to read/write data from/to an address in the protected regions, the Integrity Checker in MC will perform verification on those memory locations to detect the presence of data tampering. Verification consists of decrypting the nodes and matching them against the one stored in the tree. All tree nodes are also further recursively verified against their parent nodes until the root node. Once the data are verified, they will be moved to the cache/processor. The Integrity Checker is also responsible for updating the tree once the memory access requests have been processed. Updating involves checking if the tree needs re-balancing to move frequently accessed tree nodes closer to the root node. The detailed description of the tree construction and verification will be discussed in Section 5.2.1.

5.2 Proposed Framework

The proposed dynamic integrity tree adopts the principles of the TEC-tree [33], which provides for both data confidentiality and integrity. In this section, we describe the modifications made to the balanced TEC-tree [33] to enable dynamic skewing. As discussed earlier, the TEC-tree employs AREA (Added Redundancy Explicit Authentication) technique at the block level, where redundant data (a nonce) is added to each tree node before encryption and checked after decryption. Since TEC-tree relies on block encryption to construct the tree, it provides data confidentiality without any additional latency by

using the hardware already in place for integrity verification. This advantage makes it possible to employ the TEC-tree even for non-volatile memories (NVMs) which face stolen memory attacks due to their data remanence properties. It is worth mentioning that the proposed method for dynamically skewing the integrity tree can also be applied to other existing forms of memory integrity trees. We will introduce the new tree node specifications and its design in Section 5.2.1 and Fig. 5.2.

5.2.1 Cache-Aware Memory Integrity Tree Model

We use the term Data Chunk (DC) to refer an atomic block for authentication. To reduce the overhead of data verification, the processor should be able to easily obtain a parent's address from the node's address. Thus, the DCs have been modified to enable quick access to this information. The DC can be broadly split into two parts, 1) Plaintext or Data and 2) Nonce. The attributes of a DC is shown in Fig. 5.2. Let n_i denote the Node ID of the i^{th} node on the tree. p_i , s_i , LR_i , c_i denote the parent, sibling, side, count of write access of n_i respectively. p_i stores the node ID of the parent and s_i stores the node ID of the sibling of n_i . LR_i denotes whether the node is a left child or a right child to its parent. c_i denotes the frequency of write access made to the data stored in n_i . c_i is a local counter dedicated to a specific chunk rather than a global counter. This is to minimize counter overflows. A counter overflow will require all the encryption keys to be re-generated, hence utilizing local counters can help extend the lifetime of the encryption key.

Multiple encrypted DCs are combined to form a leaf node for the tree. The number of DCs that are grouped in a leaf node depends on the cache line size. Also, the data elements of the same leaf node are from consecutive memory addresses. During authentication, data verification is first performed on a leaf node. When there is a cache miss, the full leaf node which has a size equivalent to the cache line is transferred to the cache after

authentication. This implies that all the DCs in the leaf node have to be verified even though access is made to only one of them. Each DC in the leaf node is decrypted sequentially followed by authentication with the parent node. Note that each DC in a leaf node will have the same parent lineage since they are associated with consecutive memory addresses. As most applications exhibit locality of reference, the other DCs in the same leaf node that have been authenticated and their decrypted data elements are in the cache, will likely be accessed shortly. As such, speculative verification of the data is performed to reduce the authentication overhead. Also, as most applications display at least some spatial locality, consecutive addresses are likely to be called soon. This has been done to minimize the memory-cache access and to match the cache line size as will be discussed later.

Example As an example, we consider the tree shown in Fig. 5.2. We implement AES-256 encryption. The data and counter chunks are 256-bit blocks. In the case of DC, 128-bit data is concatenated with a 128-bit nonce. The CC has two 64-bit counters concatenated with a 128-bit nonce. We consider a cache line size of 512 bits. Thus, as can be seen in Fig. 5.2 we combine 4 data chunks to form one leaf node ($128 \text{ data bits} * 4 = 512$) so that 512 bits of verified data can be passed after one complete round of leaf to root authentication. In the event of a cache-miss, four DCs are verified, and data extracted from these nodes is transferred to the cache. As these four DCs are always transferred and evicted together to/from the cache, they share the same count and have a common parent storing their c_i 's. This format has been adopted to keep the address calculations simple keeping in mind the constraints due to the required nonce size, AES block size, and cache line size. Similarly, for various other cache line sizes, the leaf node can be formed by combining 2/4/8 data chunks, for example, if c is 1024 bit cache line then for a similar size data chunk with b as 128, we will have to combine 8 such data nodes to be processed at the same time.

Figure 5.2: Proposed dynamic skewed integrity tree structure

The verification starts by decrypting the DC to obtain the counterpart of the nonce. The counter values are also stored in the off-chip memory in a tree structure. We refer to Counter Chunk (CC) as a block where the counterpart of the nonces are stored with its nonce. The entire CC is encrypted as a single block before being stored off-chip. The layout of the counter and data chunks is shown in Fig 5.2. A block-level AREA scheme is recursively applied to subsequent levels of the tree to create CCs with redundant data being added to them. Finally, we obtain a single CC, called the root node of the tree. The root value is also stored on-chip securely.

The main objective of the recursive procedure described above, which creates the tree structure is to reduce the on-chip memory overhead to a single root node while providing full memory integrity.

5.2.2 Authentication

The authentication functions of *ReadNCheck*, and *WriteNUpdate*, are performed in a similar manner to that described in Chapter 4. Unlike the method proposed in Chapter 4, there is no set migration operation to be performed as the granularity of *ShiftUp* is now a single node. Only the re-balancing procedure is performed, which is described in the next sub-section. The idea is based on the principle that leaf nodes with a higher probability of occurrence (based on the c_i values) are shifted up closer to the root node. This is achieved in two steps, as illustrated in Fig. 4.3.

- Exchanging subtree with its uncle
- Rotation

5.3 Implementation Example

We consider an example shown in Fig. 5.3 to provide some insight into the internal working of a dynamic tree. We begin with the tree structure shown as TREE 1 with its leaf nodes placed in a skewed manner. The table below each figure describes the elements stored in that particular tree.

- TREE 1: Tree nodes 1, 2, 3, and 6 are CCs and the remaining (4, 5, 7, 8, 9) are leaf nodes comprising of multiple DCs. Node 4, 5, and 7 having a higher count value than nodes 8 and 9 are placed one level higher on the tree. We consider the following order: $9 \rightarrow 5 \rightarrow 5$ for subsequent write requests.
- TREE 2: Based on the access sequence, the next node to be accessed is 9. Thus, the count value of node 9, along with its parent, is incremented by 1. This makes it eligible to be considered for re-balancing. As the count of Node 9 is greater than its sibling's count (Node 8) by 2, and it is also greater than its uncle's count (Node 7),

Figure 5.3: Implementation example

it successfully meets the re-balancing criterion. After performing the re-balancing operation, the tree takes up the form of TREE 2 in Fig. 5.3. We can observe that Node 9 has been shifted up by one level, and Node 7 has been pushed down by one level. The re-balancing criterion is checked for Node 6 and 3 as well but is not met.

(iii) TREE 3: Next node to be accessed is Node 5. This increases the count of Node 5 to 9. Again, the re-balancing criterion is checked. As it is not yet eligible (its count is greater than it's sibling by only 1), re-balancing operation is not performed. Parent of Node 2 is also checked against the criterion, but it is not eligible.

(iv) TREE 4: Node 5 is then reaccessed, increasing it's count to 10. With this, the

re-balancing criterion for Node 5 is met. Thus after the re-balancing operation, the tree takes up the form of TREE 4 in Fig. 5.3. Node 5 has been pushed one level up to level 2, and the remaining nodes are pushed one level below. Node 4 is still on level 3. Remaining nodes i.e., 7, 8, and 9 have been pushed one level down.

5.4 Performance Evaluation

The techniques proposed in the previous chapters were evaluated only on FPGAs, which have limited memory and cache configurations. For this work, we have implemented and evaluated the framework on a system simulator, Multi2Sim. For our evaluations, we used twelve benchmarks from the SPLASH-2, PARSEC, and SPEC-CPU2006 benchmark suites on Multi2Sim [109]. The Multi2Sim simulator integrates several useful features such as separate functional and timing simulation, multi-threading and multi-processor support, and cache coherence. Multi2Sim is an application-only tool intended to simulate x-86 binary executable files. Our experiments were undertaken using a single core system, with a single fully associative cache utilizing a write-back policy and an LRU replacement policy. We evaluated the framework for six different cache configurations i.e., 256 Kb, 1Mb and 4Mb cache with 64, 128-byte cache lines. The main memory latency has been used with 100 cycles latency, 1.4 GHZ, 8B/cycle. As memory authentication contributes to significant overhead in the execution time, we measured the time required to perform integrity checking i.e., encryption/decryption and verification. In addition to comparing our work with [33], since we modify their fundamental framework, we have compared our work with ASSURE [99] to highlight the effectiveness of our work. To the best of our knowledge, [99] is the only work other than ours that tackle the problem of tree size by taking advantage of memory access patterns.

The benchmarks were simulated for three different tree architectures: 1) balanced TEC-tree (*BTEC*) [33], 2) *ASSURE* [99] that is adapted to the TEC-tree format, and 3)

proposed dynamic skewed tree (denoted as *Proposed*). Note that all the implementations are based on the TEC-tree format to ensure fairness in the evaluations. In addition, block encryption with cache line granularity is applied to all the methods for fair evaluations. AES encryption/decryption latency has been considered to be 40 clock cycles. In order to ensure fair comparisons, all the trees adopt an arity of 2.

5.4.1 Authentication Time

The authentication time is directly proportional to the number of tree levels accessed for integrity verification of a leaf node. We have measured the authentication time for six different cache configurations with varying cache size and cache line sizes. The time taken to perform the integrity check is minimum for the proposed dynamic skewed tree for all the benchmarks as shown in Fig. 5.4. The timing gain achieved varies for each application based on its memory usage pattern. From the results, it is evident that moving the nodes closer to the root reduces authentication time as both ASSURE and Proposed design performed better than BTEC. The proposed method has on average 35% lower integrity check runtime over BTEC and 18% lower runtime over ASSURE. The proposed dynamic skewed tree performs better than ASSURE due to its higher degree of adaptability where placement of each leaf node is based on the individual count value of the leaf node rather than on a MBG as in the ASSURE architecture. As expected, it can also be observed that larger cache leads to improve performance.

The benefits of skewing a tree are application dependent. In particular, the gains are mainly affected by the number of memory writes. This is due to the fact that the counter values are incremented on every write followed by a re-balancing check (see Algorithm 4.1). Thus, a large number of memory writes will lead to more frequent skewing that increases the overhead of verification.

The memory patterns of all the applications are shown in Table 5.1. The proposed method achieved high gains for applications that are sufficiently large, with a substantial

Table 5.1: Memory access patterns

Benchmarks	Reads	Writes
Blackscholes	80,242	19,549
Radix	3,212	140
Cholesky	3,418	165
FFT	15,81,279	5,42,979
Radiosity	6,777	12,05,435
Lu_cb	7,42,191	5,72,497
Raytrace	56,804,585	20,144,507
Ocean_cp	4,022	1,172
Mcf	1,30,857,287	36,484,603
Omnnetpp	1,07,159,247	69,846,205
Xalancbmk	1,77,685,047	59,298,527
Gcc	95,761,851	74,480,861

number of memory accesses. Such requests will result in trees with many levels for the BTEC implementation. The proposed dynamic skewed tree can effectively lower the overhead in verification for these applications by reducing the average number of tree levels during authentication.

We have included *mcf*, *omnnetpp*, *xalancbmk*, and *gcc* workloads from SPEC2006 in our evaluations as they have high memory traffic (≥ 1 memory access per 1000 instructions) and random data accesses. It can be observed that for these benchmarks, the proposed scheme leads to an average reduction of authentication time of about 33% and 19% over ASSURE and BTEC, respectively. Another interesting observation is applications like *Radiosity* that has a large number of memory accesses do not benefit as much from the proposed method. This is because the majority of requests are written, which are substantially higher than the reads. In this scenario, the system spends significant time performing the re-balancing operation leading to higher overhead. It is noteworthy that even with this increased overhead, the proposed method still achieves a gain of 30% over ASSURE and 25% over BTEC. The remaining three applications, *Radix*, *Cholesky*, *Ocean_cp* are relatively smaller with lesser number of accesses made to the main memory.

This results in a compact integrity tree with better cacheability of entries on-chip. If a part of the protected region is in cache, it is considered safe and hence not verified. This results in lesser number of verifications. Such non-memory intensive workloads with infrequent tree traversals have limited potential for reduction in authentication time. Thus, the benefits of skewing the tree are limited (on average 15% over ASSURE and 20% over BTEC). Among these three applications, *Radix* seems to benefit the most from the proposed method. This can be attributed to its memory access pattern, where accesses have a random nature without much temporal locality. This results in considerable memory traffic during integrity tree traversal as there is limited re-use of the cache entries. Applications like *Radix* can attain immense benefits from dynamically skewed trees due to the reduced number of authentication levels. It can be observed that *Radix* achieves on average a gain of about 38% over BTEC.

Finally, Fig. 5.4 shows the effect of varying the cache size and the block size on the performance. In general, a larger cache reduces the number of off-chip accesses and overall execution time of applications. Having a more significant cache line also reduces the overhead of memory verification due to the reduction in the levels of the integrity tree.

5.4.2 Energy

We perform energy evaluations using McPAT (Multicore Power, Area, and Timing) [110]. McPAT is an integrated power, area, and timing modeling framework for multithreaded, multicore, and manycore architectures. The simulations are performed using the same system configurations and technology as Multi2Sim.

The results in Table 5.2 and 5.3 compare the total runtime and energy consumption of the eight applications when they are executed using the BTEC and proposed method for memory authentication. Each row in the tables shows the runtime and energy consumption for the BTEC running on a system with a larger cache size compared to the proposed

(a)

(b)

(c)

(d)

(e)

(f)

(g)

(h)

Figure 5.4: Authentication time (in ms) for twelve different applications with varying cache configurations

method. For example, in Table 5.2, the BTEC method is executed on a system with 4MB cache while the proposed method is executed on a system with 1MB cache. Similarly, Table 5.3 compares the runtime and energy consumption when the BTEC is executed on a system with 1MB cache while the proposed method executes on a system with only 256KB cache. The 4th and 7th column of the tables report the percentage reduction in runtime and the percentage energy savings of the proposed method over BTEC. It can be observed that for all the applications considered, the proposed method has almost similar ($\leq 6\%$ decrease) or better performance over BTEC even though the former is executed on a system with significantly smaller cache. For *Radix* and *Raytrace*, the proposed method achieve almost 30% and 15% lower runtime respectively. For all the cases considered, the proposed method leads to energy reduction due to the use of a smaller cache. This reduction can be over 43% in certain applications. Similar results are also observed when the proposed method is compared with ASSURE. Although the gains are not as significant, the proposed dynamic skewed tree was still able to achieve energy savings at the cost of slight or no performance loss.

The combined advantages of the proposed integrity tree structure (which was designed in a cache-aware manner) and our proposed mechanism to dynamically skew the tree, result in comparable performance with existing memory authentication methods that require almost double the cache size. Hence, our memory authentication method provides the advantage of choosing systems with smaller caches to minimize energy consumption without compromising performance.

5.4.3 Instruction Per Cycle (IPC)

Fig. 5.5 shows the impact of memory verification on the overall application performance in terms of IPC for a system with 256 KB cache and 64 Byte cache line. For all the different benchmarks, the IPC (instruction per cycle) of the three schemes are shown

Benchmarks	Execution Time (sec)		% Difference	Energy (Joules)		% Energy Saved
	4MB BTEC	1MB Proposed		4MB BTEC	1MB Proposed	
Blackscholes	647.85	677.23	-4.5%	3368.85	3047.55	9.57%
Radix	4.91	3.21	+34.5%	25.54	14.46	43.3%
Cholesky	0.62	0.63	-1.08%	3.27	2.86	12.52%
FFT	1060.10	1065.14	-0.47%	5512.57	4793.18	13.05%
Radiosity	229.8	235.38	-2.42%	1195.00	1059.24	11.36%
lu_cb	1750.83	1758.8	-0.48%	9100.71	7913.60	13.04%
Raytrace	3114.39	2550.99	+18.09%	16195.87	11479.46	29.11%
Ocean_cp	1.31	1.34	-2.23%	6.84	6.05	11.52%
Mcf	387.58	606.24	-4.8%	2015.43	1828.11	9.29%
Omnetpp	778.48	780.47	-0.25%	4048.13	3512.14	13.24%
Xalancbmk	1128.32	1152.85	-2.17%	5867.30	5187.86	11.58%
Mcf	1202.59	1237.25	-2.88%	6253.51	5567.66	10.96%

Table 5.2: 4MB Balanced vs 1MB Proposed

Benchmarks	Execution Time (sec)		% Difference	Energy (Joules)		% Energy Saved
	1MB BTEC	256KB Proposed		1MB BTEC	256KB Proposed	
Blackscholes	671.30	705.49	-5.09%	3067.86	2941.93	4.10%
Radix	8.55	5.73	+32.9%	39.08	23.91	38.8%
Cholesky	0.96	1.00	-4.07%	4.31	4.17	5.02%
FFT	1096.98	1108.04	-1.00%	5013.20	4620.58	7.83%
Radiosity	363.13	364.98	-0.50%	1659.53	1521.97	8.28%
lu_cb	1936.06	1922.24	+0.71%	8847.83	8015.77	9.40%
Raytrace	3753.56	3239.53	+13.69%	17153.79	13509.52	21.24%
Ocean_cp	1.46	1.56	-6.43%	6.69	6.50	2.88%
Mcf	446.58	451.58	-1.11%	2009.62	1883.10	6.29%
Omnetpp	799.46	796.84	0.32%	3597.59	3322.83	7.63%
Xalancbmk	1183.69	1197.83	-1.19%	5326.62	4994.97%	6.22%
Gcc	1259.46	1262.58	-0.24%	5667.59	5264.97	7.10%

Table 5.3: 1MB Balanced vs 256KB Proposed

Figure 5.5: Instruction per cycle comparison

with AES implemented as custom instructions. It can be observed that the proposed method achieves an average IPC improvement of 13% over the balance tree and 10% over ASSURE. The maximum gain is close to 50% for Radix. Smaller applications like Ocean_cp show minimum gain.

5.5 System Overhead

The following are the memory and performance overheads of the proposed scheme:

- (i) Memory Overhead: Our baseline TEC-tree concatenates a nonce along with data bits and encrypts them together as a single block. This block-level encryption mechanism provides confidentiality of data at no additional cost. Nonce values cause the extra memory overhead for the tree. As the trees are restructured dynamically, we add attributes to the nonce to enable easy retrieval of tree locations from memory.

The memory overhead for the baseline TEC-tree and the proposed method are as follows:

$$O_{BTEC} = \frac{l_p + n_{BTEC}}{l_p(A - 1)} \quad (\text{Eq. 5.1})$$

$$O_{Prop} = \frac{l_p + n_{Prop}}{l_p(A - 1)} \quad (\text{Eq. 5.2})$$

where l_p is the data bits in the data chunk, n are the nonce bits and A is arity.

The nonce size differs for both methods by $2 \cdot \log_2 N$ bits due to the additional attributes of *parent* and *sibling* required for the proposed method as discussed earlier.

$$n_{BTEC} = count + \log_2 N \quad (\text{Eq. 5.3})$$

$$n_{Prop} = count + 3 * (\log_2 N) \quad (\text{Eq. 5.4})$$

where N is the maximum number of tree-nodes and *count* is counter value size.

All integrity tree designs will typically incur additional memory overhead (about 1x to 2x times of the actual data protected). Although the proposed method incurs slightly higher memory overhead than the baseline to facilitate fast tree traversal, the cost of this additional storage does not impose a problem since the content of any integrity tree (i.e., MAC, hashes, encryption counters) is always stored in external memory.

- (ii) Update (Shift-Up) Procedure: For every write request on a memory address protected by the tree, an update procedure had to be performed recursively on all nodes from the leaf node till the root. This recursive checking adds to the overall time required for authentication. In spite of this additional overhead, we can achieve performance gain over the balanced tree. This can be attributed to the fact that verification in itself is a recursive process, where each node on the path from data to the root has to be verified. The verification process includes decrypting/encrypting the tree nodes, which is computationally intensive. The relative overhead caused by the update procedure is small as it only involves changing pointers for the node attributes. Hence, we can achieve the reported performance gains despite this overhead.

5.6 Summary

In this chapter, we presented a memory authentication framework for a dynamically skewed integrity tree which combines architecture-specific optimization of the integrity tree structure with runtime mechanisms to minimize the authentication overhead. A new node structure is proposed that enables the integrity tree to be restructured dynamically with low-performance overhead and single node granularity. Experiments show that the proposed method achieves the highest performance gains over existing methods for reasonably large applications where the majority of the memory accesses are read operations. We also show that the proposed framework can select a suitable cache configuration to minimize energy consumption. The next chapter will discuss techniques to improve the scalability of the proposed dynamic skewed memory integrity tree for applications with large memory accesses. We will also present the security analysis of the proposed dynamic skewed tree in the next chapter.

Chapter 6

Performance Optimizations and Security Analysis

In this chapter, we will propose three techniques to enhance the performance of the dynamic skewed memory integrity tree. First, we investigate the benefits of incorporating a dedicated Tree-Cache to cache the nodes of the integrity tree. We propose to initialize the tree structure using van Embde Boas (vEB) organization to improve cache efficiency. Second, we propose a Skew-and-Split tree structure which makes it possible to increase tree arity by reducing counter sizes without incurring counter overflows¹. Finally, we propose to implement the AES module using custom instruction for accelerating the encryption/decryption steps during memory authentication. The security analysis of the proposed methods in this thesis will also be discussed.

6.1 Dynamic Skewed Memory Integrity Tree with Tree-Cache

The memory contents in an integrity tree are always stored as the leaf nodes of the tree after encryption/hashing. A hierarchical tree structure is built on top of the leaf nodes by recursively applying a primitive authentication technique (e.g., MAC, hash). Authentication of a leaf node would involve verification of each node against its parent

¹This work has been published in [4]

node, from the leaf up till the root of the tree. A copy of the root, stored on-chip, is assumed to be safe from attack. After the final step of successfully verifying the root node with its copy stored on-chip, data is considered safe and forwarded to the data cache. The whole process, which entails verification at each level of the tree, increases the execution time substantially. To reduce this overhead, a dedicated Tree-Cache can be used for caching the integrity tree [111]. For such a system, the verification stops as soon as a node of the tree that is already cached is encountered. That node then behaves as a local root and is assumed to be safe. However, such approaches may lead to overall performance degradation due to cache contention. Using a large Tree-Cache to reduce the cache misses of the tree nodes also becomes prohibitive for low-cost embedded systems.

In this section, we propose a cache-aware memory integrity tree organization to take advantage of a dedicated Tree-Cache in order to reduce the memory bandwidth for verification. To further reduce the memory authentication overhead, we present a method called Cache-Aware Dynamic Skewed Tree (CADST) that combines the proposed memory layout with the dynamic skewed memory integrity tree framework presented in the previous chapter.

6.1.1 System Overview

In Fig. 6.1, we present the security design model. The central element of the security model is a memory controller (MC) which is responsible for the authentication process. It consists of the following components:

- (i) Encryption/Decryption Unit: We employ AES-128 as the encryption standard.
- (ii) Integrity Checker (IC): Each tree node is matched to its parent node. In case of a mismatch, an alarm is raised.

Figure 6.1: Security design overview

- (iii) Root Node: A copy of the root node of the tree is stored on-chip and is assumed to be safe. The final requirement for data to be considered safe is that the root node of the tree must match with the root value stored on-chip.

The process of authentication begins when a read/write request is sent to an address in the Protected Data region of the external memory.

Step 1: The leaf node corresponding to that specific address is forwarded to the AES unit for decryption and followed by verification.

Step 2: For verification, a request to access the parent node is initiated by the MC.

The latter can lead to the following outcomes:

- Request sent to Tree-Cache is met with a cache hit: Tree-Cache stores the decrypted data. Thus the requested node value can be directly sent to the Integrity Checker for matching. If matched, verification stop and the data is forwarded.
- Request sent to Tree-Cache is met with a cache miss: The node is then forwarded from the Integrity Tree region of the external memory to the AES unit for decryption. Once decrypted, it is then sent to the Integrity Checker to be matched with its child node.

Step 2 is performed recursively in case of a miss, until the root node of the tree is reached or a cache hit occurs. The root node is compared against the value stored on-chip, and if they match, data is passed to the data cache for processing. In case of a write request, MC will initiate a command to restructure the tree.

6.1.2 Proposed CADST Framework

In integrity trees employing AES block encryption, continuous memory addresses are segregated into blocks, and a fixed number of blocks are combined to meet the CL size. These blocks are encrypted/decrypted together. A data chunk is brought on-chip with its siblings in the tree that belong to the same cache line after verification. Thus to design integrity trees with a dedicated Tree-Cache, the most straight forward approach will be to create tree nodes with the same or a direct multiple size of a single cache line (CL). For example, in SGX, the granularity of MEE is fixed to 512 bits (same as CL size), and the MEE tree data structure is designed based on this constraint.

6.1.2.1 Initializing Tree Structure with Cache-Aware Organization

A binary tree of height, h , equal to $\log N$, is initialized to secure N elements. Conceptually, the edges at the middle level of the tree are split, between nodes of height $h/2$ and $h/2 + 1$. This partitions the tree into a top recursive sub-tree A of height $h/2$, and several bottom recursive sub-trees $B_1, \dots, B_k$ of height $h/2$. If all non-leaf nodes have the same number of children, then there will be $\sqrt{N}$ recursive sub-trees at the bottom, each with roughly $\sqrt{N}$ nodes. After splitting, the upper sub-tree elements are grouped followed by lower sub-tree elements. The vEB layout is recursively applied to each sub-tree. At each step of recursion, the size of the sub-tree being grouped is the square root of the size of the sub-trees in the previous step. Consequently, at some point, a single sub-tree group can be retrieved in a single memory transfer, and this final partition will create a tree with sub-trees of size equivalent to a single CL.

Figure 6.2: Standard memory integrity tree structure

The vEB version of the standard memory integrity tree in Fig. 6.2 is shown in Fig. 6.3. One can observe how the node ID of the CCs in Fig. 6.3 are different where the integrity tree was initialized with the vEB layout. For Fig. 6.3 both child and parent node will be cached in the same line whereas for Fig 6.2 a cache line comprises of node along with its siblings. This vEB organisation will reduce the number of cache misses as during verification, the node access is always followed by its parent node access (as will also be explained later in Section 6.1.3.3). Note that the proposed layout of the memory integrity tree is cache-oblivious, except for the requirement that each sub-tree is aligned with the cache block boundaries.

6.1.3 Proposed Skewing Framework

6.1.3.1 Overview

A skewing operation of *ShiftUp* is performed after write requests. For that, the c_i value of a given chunk is compared to its neighborhood values. If a chunk has higher c_i value than its immediate neighbors, then it should be shifted one level up, closer to the root node. In this manner, we are able to reduce the height difference between the node and the

Figure 6.3: Proposed vEB memory integrity tree structure

root. This reduces the number of verifications to be performed. The *ShiftUp* operation is performed before nodes are cached so that upon eviction, they are placed at their most recent position. A skewed tree can be seen in Fig 6.4. The benefits of dynamically skewing of a tree have been extensively discussed in Chapter 5.

6.1.3.2 Proposed Dynamic Skewing

The vEB memory layout can help achieve minimal cache misses if the proposed *ShiftUp* does not shuffle nodes forming a sub-tree. This is made possible by performing the restructuring operation on sub-trees and not single tree nodes i.e., *ShiftUpSubTree*.

ShiftUpSubTree: Let T be the sub-tree to be checked for shifting. Further, let Q be its parent sub-tree. We should note here that the count of a sub-tree is equal to the count of its root node.

Step 1: Check for the rebalance criterion i.e., if c_i of T is greater than its corresponding

sibling sub-tree count, c_j by 1, and is also greater than its uncle sub-tree count c_k , then,

Step 2: Exchange T with its uncle sub-tree.

Step 3: Exchange the new children sub-trees of Q.

Step 4: Update the counter value of all the nodes of Q.

Step 5: Recursively perform Steps 2 to 4 for all sub-trees on the path from T to the final sub-tree with the root node.

Figure 6.4: Skewed tree with sub-trees of size less than CL

The re-balancing process and verification process happen concurrently. The decrypted counter chunks are checked for re-balancing during the verification process. All changes are made prior to re-encrypting the nodes at the end of the verification process.

6.1.3.3 Memory Authentication

To avoid cache contention, we use a separate Tree-Cache on-chip to store the tree meta-data. The authentication functions of *ReadNCheck* and *WriteNUpdate* are now conducted as follows:

ReadNCheck: This function is called when there is a read request sent to the Protected Data region that has not been cached and thus requires verification.

- (i) Call the requested DC's parent CC. If it is cached in the Tree-Cache, return the cached data for matching. The process completes, otherwise
- (ii) Fetch CC from external memory.
- (iii) Forward decrypted CC to IC for verification. If verified, place the contents of the child chunk into the Tree-Cache and move on to verify the parent chunk (note that this step only applies when a CC is being verified).
- (iv) Repeat Step 1 until a cache hit is encountered or the root node is reached.
- (v) If root node verifies correctly, return the requested data.

WriteNUpdate: This function is called when there is a write request sent to the Protected Data region that is not cached and thus requires verification.

- (i) Call the requested DC's parent CC. If it is cached, return the cached data for matching. Also, increment the c_i value. The process completes, otherwise
- (ii) Fetch CC from external memory, increment c_i .
- (iii) Check for *ShiftUpSubTree*.
- (iv) Place modified decrypted CC in Tree-Cache and forward it to IC for verification.
- (v) Repeat Step 1 until a cache hit is encountered or the root node is reached.
- (vi) If root node verifies correctly, return the requested data.

WriteBack: This function is performed upon eviction of a dirty CC from the Tree-Cache.

- (i) Update the CC in the external memory.
- (ii) Recursively update the parent CCs, if not cached, up till the root with the new c_i values.

Table 6.1: Architectural parameters used in simulations

Architectural Parameters	Specifications
Clock Frequency	1 GHz
Data Cache	256 KB, 1-way
Tree-Cache	16KB/8KB, fully associative
Cache Latency	4
Replacement Policy	LRU
Write Policy	Write Back
External Memory Latency	100
AES Latency	40

6.1.4 Performance Evaluation

For our evaluations, we simulated applications from SPEC-CPU2006, SPLASH-2, and PARSEC benchmark suites on Multi2Sim [109]. Multi2Sim is an application-only tool intended to simulate x86 binary executable files. Our experiments were run on a single core system. The architectural parameters are listed in Table 6.1. We ran the benchmarks for four different tree architectures with a dedicated Tree-Cache: 1) conventional balanced TEC tree (BTEC) [33], 2) cache-aware balanced TEC tree (CABTEC) 3) dynamic skewed tree (DST) [2], 4) proposed cache-aware dynamic skewed tree (CADST). BTEC [33] is a balanced integrity tree employing block-level AREA encryption, as described in Chapter 3. CABTEC is the BTEC tree implemented with the vEB layout in memory, as explained in Section 6.1.2.1. DST [2] skews the tree nodes depending on the frequency of access with the most frequently access closer to the root. The DST implementation does not employ the vEB memory layout. CADST is the proposed method where we initialize the DST with the vEB layout and use *ShiftUpSubTree* to skew the tree during runtime. We have presented results for four different Tree-Cache configurations, i.e., 16Kb and 8Kb size with 128, 64-byte cache lines.

Fig. 6.5 compares the normalized performance of all four models. The performance metric is based on the runtime required to perform the memory authentication. It is

evident that dynamic skewing improves runtime performance. DST has on average 20% lower runtime compared to the baseline BTEC model. Further, it can be observed that incorporating a cache-aware memory layout for the balanced integrity tree (CABTEC) can reduce the authentication time by approximately 14% compared to BTEC. By introducing the dynamic skewed tree mechanism which leverages the cache-aware memory layout, the proposed CADST model performs the best amongst the four models, achieving an average speedup of approximately 15%, 17%, and 25% over DST, CABTEC and BTEC respectively.

The benefits of the proposed method depends on the application and its workload. The height of the balanced integrity trees, shown in Table 6.2, is proportional to the size of the Protected Data. Large trees result in a lot of storage and runtime overhead, and hence they have a higher potential for improvement. Also, for memory-intensive applications such as Lu_cb, Mcf, Omnetpp, Xalancbm, and Gcc (≥ 1 memory access per 1000 instructions), the benefits of skewing are more pronounced. Fig. 6.5 shows the effects of varying the Tree-Cache size and the block size on the performance. In general, a larger cache reduces the number of off-chip accesses and overall execution time of applications.

Another factor that directly affects the authentication time is the number of times the average number of CCs are accessed per DC. As such, we analyzed the memory traffic bottlenecks due to the CC accesses, which is the main cause of overhead. Fig. 6.6 shows the memory traffic bloat (counter accesses per data access) for all the models. This metric is directly proportional to the number of tree levels a leaf node will access while being authenticated. The benefits of skewing can also be observed here. Moreover, CADST being more cache-friendly makes the least accesses from the external memory for the CCs of all the applications. The improved performance due to the vEB layout is also highlighted in the difference between BTEC and CABTEC models.

Table 6.2: Height of memory integrity trees

<i>Benchmark</i>	Blackscholes	Radix	Cholesky	FFT	Radiosity	Lu.cb
<i>Height of tree</i>	13	23	11	13	16	12

<i>Benchmark</i>	Raytrace	Ocean_cp	Mcf	Omnetpp	Xalancbmk	Gcc
<i>Height of Tree</i>	22	11	27	16	23	24

6.2 DISSECT: Dynamic Skew-and-Split Tree

Another technique for reducing tree height is by increasing arity. To construct shallow trees with larger arity, [97, 98] employ varying encryption counter configurations. Such approaches, however, have their limitations. Firstly, as reported in [97], increasing the tree arity is not a strategy that can be adopted universally. For example, they cannot be applied to MAC trees whose arity is limited to 8 irrespective of the counter design in the tree node. The arity for MAC trees depends on the Message Authentication Codes (MAC) per entry and the cache line size. Secondly, the success of methods such as [97, 98] is dependent on the counter nodes being cached efficiently, as this contributes to reducing the number of counter writes and overflows on higher levels of the tree. However, introducing a dedicated cache for the counter nodes is not always possible, especially for embedded systems with tight constraints.

Moreover, approaches that rely on caching the tree nodes have reported overall performance degradation due to cache contention [102]. Storing the metadata of the tree nodes in caches for large trees consumes substantial memory space. [102] reported that unless the cache is reasonably big, the tree nodes will occupy 25% - 50% of the cache, leading to a large number of cache misses.

When a counter overflows, it is essential that a new key is generated and the entire tree is re-encrypted using a new key. Without this preventive measure, the tree contents will not be secure against replay attacks if counters are re-initialized. As such, the use of larger arity can inadvertently lead to higher performance overheads due to more frequent

counter overflows. For example, the technique proposed in [97] employs a 128-ary integrity tree which is made possible through the use of a single sizable major counter per cache line and several smaller minor counters in the tree nodes. As mentioned in their work, a non-optimized design packing of 128 counters per cache line results in 3-bit minor counters that can overflow in just 8 writes. As each overflow requires 256 extra memory accesses, this will result in significant performance degradation. Even with their optimized design, the counters tend to overflow frequently, especially for specific data streaming applications, leading to high-performance overheads. The experiments conducted in [97] demonstrated cases where the higher performance overheads due to counter overflow offset the gains obtained due to the reduction in tree height. The counter overflow problem has a more significant impact on Non-Volatile Memories (NVM) [112], since writing to NVM cells is much slower and results in considerable power consumption.

Reducing counter size to increase arity, reduces the security against replay attacks. The probability of a successful replay attack depends directly on the size of the counter. If an attacker manages to guess the counter value, the system security will be compromised. Thus, increasing arity by reducing counter size is only an option for systems that do not require a high level of protective measure [113], or if other mechanisms for detecting replay attacks are available.

In this section, we propose a memory integrity tree structure and the corresponding authentication framework that simultaneously addresses the problems associated with tree height and counter overflow. We call the proposed framework DISSECT, which stands for Dynamic Skew-and-Split Tree for Memory Authentication. DISSECT enables the height and arity of the proposed memory integrity tree structure to be dynamically adjusted based on the run-time memory access patterns to reduce the performance overhead for authentication. This is achieved by 1) initializing an integrity tree structure with the largest arity possible to meet the security requirements, 2) dynamically skewing the tree

such that the more frequently accessed memory locations are positioned closer to the tree root, and 3) dynamically splitting the tree at nodes with counters that are about to overflow.

DISSECT leverages the complementary benefits of reducing tree height and increasing arity. This is based on the observation that nodes which tend to encounter counter overflow are also the nodes that are most frequently accessed (the counters associated with each memory location are incremented on the memory write access). DISSECT will overcome the tree height problem by shifting the nodes that are associated with the most frequently accessed memory locations closer to the root (to reduce the number of authentication levels of these nodes), and restrict the splitting process to these nodes when their counters are about to overflow.

(a)

(b)

(c)

(d)

Figure 6.5: Authentication time with a) 16KB Tree-Cache, 64B cache line, b) 16KB Tree-Cache, 128B cache line, c) 8KB Tree-Cache, 64B cache line, d) 8KB Tree-Cache, 128B cache line

Figure 6.6: CCs per DC accessed

Figure 6.7: An example structure of (a) 4-ary DISSECT, (b) Post skew, (c) Post skew-and-split

6.2.1 Proposed Tree Structure for DISSECT

Our framework begins with a balanced integrity tree with a -ary. An example of a 4-ary tree is shown in Fig. 6.7a. In DISSECT, initially, a balanced tree is divided into different groups with each of them comprising of $a/2$ nodes. The numbers in Fig. 6.7a indicate the group numbers. The tree arity is determined based on the system's cache and security requirements, i.e., cache line size and encryption block. At runtime, the tree is skewed according to the memory access patterns. We have magnified the left portion of the 4-ary tree in Fig. 6.7b (i.e., the red bounding box in Fig. 6.7a) to illustrate an example of skewing. In this example, Group 15 has been switched with its uncle group, Group 3, since the former was accessed more frequently. As a result, Group 15 has been elevated one level closer to the root. Subsequent access to Group 15 will require a lesser number of authentication steps.

Furthermore, when any node is about to exceed its maximum local counter value, the node is split. A new node is created, and half of its children are assigned to the new node. An example of the tree split is shown in Fig. 6.7c. We assume that the local counter of node 1 has reached its maximum value. Consequently, the arity of node 1 has decreased from 4 to 3, resulting in extra counter bits for nodes in Group 15. We will discuss in detail each of the above steps in the following sections.

Figure 6.8: (a) Proposed node structure, (b) Memory block with 512-bit cache line for TEC-tree, and (c) Memory block of DISSECT

To achieve dynamic skewing and to split efficiently, we need to redesign the nodes of the memory integrity tree. There is an additional attribute of *Split* that is also stored in the CC as can be seen in Fig. 6.8a. *Split* denotes whether the node or any of its children have split and thus have an increased counter size. A *Split* flag is necessary for checking a given node's counter width.

6.2.2 Proposed Skew-and-Split Framework

The proposed framework consists of the following steps: 1) initializing an integrity tree structure with the largest arity possible to meet the security requirements, 2) dynamically skewing the tree such that the more frequently accessed memory locations are positioned closer to the tree root, and 3) dynamically splitting the tree at nodes with counters that are about to overflow.

6.2.2.1 Initialize Tree Structure

A memory block (data + nonce) loaded by the processor on a cache miss must be a multiple of the ciphered-block length. Therefore, the 'data' width for a given tree node is

determined such that on a cache miss, a single memory block unit will forward 'data' of size equal to the cache line size upon verification. The structure for a 512-bit cache line with 256 cipher block length is shown in Fig. 6.8b. Based on Eq. 6.1, we can determine the arity and 'counter', where 'counter' is the bit width of a single local counter of each tree node.

$$arity = \frac{data}{counter} \quad (\text{Eq. 6.1})$$

We also improvised the structure of the DC introduced in Fig. 6.8a and split the additional attributes (i.e. p_i , s_i , LR_i) over the data chunks in a single memory block as shown in Fig. 6.8c to maximize the memory utilization. This is possible as on a cache miss, these data chunks of a single memory block have to be verified together. As such, the proposed structure avoids the redundancy of storing the parent/sibling information in each node individually.

As an example, for a 512-bit cache line, the minimum size of the nonce required for data to be equal to the cache line is one AES block (256-bit). The resulting memory block distribution is three AES blocks long, i.e., 768-bit, as shown in Fig. 6.8b. To maintain 170-bit 'data', we calculate the 'counter' size to extend the arity to 8 using Eq. 6.1, which is 21-bit. The possibility that a replay attack will be successful is $1/2^{21}$. Following the same principle, we customize data/counter chunks of the proposed tree structure to fit the given cache line, as shown in Fig. 6.8b and 6.8c. Note that the counter width should be determined based on the security requirements against replay attacks, which is discussed in Section 6.4. Compared to the TEC-tree with the same security strength, the proposed method results in a lower arity. To retain the 21-bit width of the counters, the proposed tree will only be able to accommodate six child counters instead of eight. Similarly, different counter sizes will result in different arity. For the TEC-tree, arity must be a power of 2 to simplify the hardware computations of parent address [33]. Such a

restriction does not apply to the proposed design as the parent node number is explicitly stored in all the chunks.

The extra attributes (i.e., p_i , s_i , LR_i , and $Split_i$) of each CC utilize extra bits, reducing the remaining bits to store counter values. For a 32-bit tree, the last level is made up of DCs and cannot be a parent to any node. Thus, p_i requires 31-bits. s_i stores sibling group number (each group has $a/2$ nodes, so there are half as many groups as parents) and requires 30-bits. $Split_i$ will hold information required to decide if a split has occurred, and if so, which of the left or right group of children was assigned to new nodes. LR_i indicates if the node is a left or right child to its parent.

6.2.2.2 Dynamic Skewing

For an a -ary integrity tree, the attributes must now incorporate the information of $(a-1)$ siblings instead of just 1. To avoid storing this additional meta-information, we create 'group' of $a/2$ nodes and store a group ID as s_i . We have already discussed the benefits of dynamically skewing a tree extensively in the previous chapters. We have now improvised the *ShiftUp* algorithm introduced in Chapter 4 for higher arity trees. The *ShiftUpGroup* procedure to perform dynamic skewing is applied to such groups instead of single nodes. The storage is then reduced to storing only a single sibling group number (similar to a 2-ary design where a single sibling node number is stored).

ShiftUpGroup: Let T be the group whose node is to be checked for shifting. And let Q be the parent node.

- (i) Check if total c_i of T is greater than its corresponding sibling group, s_i , total count by 1, and is also greater than the total count of its uncle group, s_{p_i} .
- (ii) Exchange T with its uncle group.
- (iii) Exchange, the new children node of Q , i.e., switch their positions.
- (iv) Recursively perform steps 2-4 for all nodes on the path from T to the root node.

6.2.2.3 Dynamic Splitting

Algorithm 6.1: Dynamic Splitting

```
A, Q : pointers to Nodes
while A is not root do
  if  $count_{p_A} = max$  then
    Q  $\leftarrow$  create new node
     $p_Q \leftarrow p_A$ 
     $s_{s_A} \leftarrow node_Q$ 
     $s_Q \leftarrow s_A$ 
    if  $\sum_{i=my\_group} c_i \geq \sum_{i=sibling\_group} c_i$  then
      for all  $n_i$  in my group do
         $p_i \leftarrow Q$ ;
        Update  $s_i$ ;
      end
      Update  $count_{Q,A}$ 
    else
      for all  $n_i$  in my sibling group do
         $p_i \leftarrow Q$ ;
        Update  $s_i$ ;
      end
      Update  $count_{Q,A}$ 
    end
  else
    A  $\leftarrow P_A$ 
  end
end
```

As shown in Algorithm 6.1, when a counter value reaches its (max-1) permissible limit, it is selected for splitting its children nodes. A new node is created, and half the children are assigned to this node, thereby providing additional bits to their respective counter values to prevent overflowing. It is noteworthy that although splitting will push nodes one level down, the subsequent shift operation promotes the nodes to a higher level. Also, splitting is performed recursively, so it is highly likely that further splits will follow on the same nodes. As each division incurs an additional tree level, the tree size is bound to increase. However, the number of increased levels will be relatively small compared to

the original tree height. This is because nodes which undergo splitting are likely to have shifted closer to the root (since they are accessed frequently).

6.2.3 Performance Evaluation

Figure 6.9: Analysis of counter overflow problem

For our evaluations, we simulated applications from SPEC-CPU2006, SPLASH-2, and PARSEC benchmark suites on Multi2Sim [109]. Multi2Sim is an application-only tool intended to simulate x86 binary executable files. We perform experiments on a single-core system. We ran the benchmarks for two different tree architectures: 1) balanced TEC-tree implementation (BTEC) [33], and 2) DISSECT. Similar to the example explained in Section 6.2.2.1, we assume a system with 512 cache line.

In order to understand the counter overflow problem in existing methods, we first show the authentication time for write-intensive applications, which have a high probability of overflowing, (i.e., *Raytrace*, *Gcc*, and *Omnetpp*) implemented with BTEC in Fig. 6.9. Authentication time is the time spent by the processor to perform integrity check on the leaf nodes. The authentication time is directly proportional to the number of tree levels accessed during integrity check. The counter width is varied from 26 to 21. Based

Figure 6.10: Normalized authentication time

on Eq. 6.1, when the counter width is reduced to 21, it is possible to increase arity to 8. We can observe that the authentication time increase as the counter size decreases. With a single bit reduction in the counter width, the authentication time will increase exponentially for certain applications. In scenarios where the counter overflows, the cost of re-encrypting the entire tree will depend on the data size being protected. *Gcc* has the largest tree size in terms of data region being protected. Thus, the overhead due to overflow for *Gcc* is more pronounced. For *Omnetpp*, although it overflows, the overhead is not significant as only a relatively smaller data size needs to be protected. For *Raytrace*, we can observe that increasing arity is still able to improve performance despite counter overflows. This is because *Raytrace* overflows the least amount of time compared to the other applications in Fig. 6.9.

We have presented results in Fig. 6.10 to compare the normalized authentication time. For practical scenarios, a 32-bit counter is a safe size. Hence, we start with a 2-ary BTEC and 2-ary DISSECT with 32-bit counters; and then increase the arity to 8-ary BTEC and 6-ary DISSECT with smaller 21-bit counters. Note that the security strength for the 2-ary BTEC and 2-ary DISSECT, as well as the 8-ary BTEC and 6-ary DISSECT are the same respectively. It can be observed that for the same security strength,

DISSECT has significantly lower authentication overhead than BTEC. 6-ary DISSECT, even with a lesser arity than 8-ary BTEC, shows on average 20% improvement in terms of reduction in authentication time over BTEC. For memory-intensive applications (e.g., *Omnetpp*, *Xalankbmk*, *Gcc*) the counter overflow overhead for 8-ary BTEC actually causes the applications to perform even worse than the same tree with 2-arity. The results demonstrate that for the same security strength, DISSECT will outperform BTEC with a higher arity tree, as it is capable of reducing the overhead due to counter overflow.

6.3 Custom Instructions

As part of our future work, we plan to develop a programmable memory controller that consists of processors with custom ISAs that can be adapted to the memory workloads in order to further reduce the performance overhead. A similar work to this is reported in [114]. In this section, we will evaluate the benefits of integrating a custom hardware AES module that is tightly coupled with the processor in the memory controller in order to accelerate the encryption/decryption steps on memory authentication.

6.3.1 AES Algorithm

The AES algorithm is based on a combination of substitution and permutation, and is a variant of the Rijndael family of ciphers. AES uses a fixed block size of 128 bits, along with a key size of either 128, 192 or 256 bits [115]. AES operates on 4x4 column-major order matrix of bytes, called the state. A 128-bit key requires 10 cycles of repetition of transformation rounds to convert the plain text to the cipher text. The flowchart for the AES algorithm is shown below.

- Key Expansion: The last four bytes of the key will undergo a G function. It simply performs substitution byte and the addition of $rconk$ value, where k represents the current round. The output will then be added to the first four bytes to generate

Figure 6.11: Flowchart for a generic AES algorithm for encryption

the new subkey. Successively, the newly generated bytes will be added to the next four bytes of the old subkey to produce the rest of the new subkey.

- **AddRoundKey:** The 128-bits data will be XOR with 128-bits subkeys generated by the key expansion.
- **SubBytes:** Each byte is mapped to another byte using the Rijndael S-box. The mapping provides randomness in the output to get around attacks that exploit algebraic properties. An inverse S-box is used during decryption to reverse the mapping operation.
- **ShiftRows:** The 128-bits data is divided into bytes. Excluding the first row, the remaining row will shift left by one, two, and three positions. The shifting operation forces each block to be dependent on all other blocks, such that a change in any block will affect the remaining blocks. During decryption, the rows will be shift right to reverse the operation.

Figure 6.12: Key Expansion

- MixColumns : Each column will be multiplied with a matrix. Multiplication will result in each byte exceeding polynomial of order x^7 which has to reduce using the irreducible polynomial $x^8 + x^4 + x^3 + x + 1$.

The AES algorithm serves as the memory encryption/decryption standard for our analysis, where each of the memory blocks is encrypted using AES. To reduce the

Figure 6.13: Substitution Byte

Figure 6.14: Shift Row

$$\begin{bmatrix} W_{00} & W_{10} & W_{20} & W_{30} \\ W_{01} & W_{11} & W_{21} & W_{31} \\ W_{02} & W_{12} & W_{22} & W_{32} \\ W_{03} & W_{13} & W_{23} & W_{33} \end{bmatrix} \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} W_i \\ W_{i1} \\ W_{i2} \\ W_{i3} \end{bmatrix} = \begin{bmatrix} W'_i \\ W'_{i1} \\ W'_{i2} \\ W'_{i3} \end{bmatrix}$$

Figure 6.15: Mix Column

latency of encryption/decryption during memory integrity verification, we evaluate the performance benefits of implementing the 128-bit mix-column AES algorithm as a custom instruction [116]. Custom instructions (CI) enable the acceleration of time-critical

software algorithms by introducing custom hardware blocks that are tightly-coupled to the Arithmetic Logic Unit (ALU) of the processor.

The speedup with the proposed custom instructions is obtained through realistic assumptions in the simulations on Multi2Sim. In order to achieve this, we first determined the latency of the software implementation of AES algorithm in Multi2Sim, which is 40 clock cycles. The full AES algorithm was then implemented as a 5-stage hardware unit and we obtained the corresponding hardware synthesis results in Synopsys (using the same target technology process as in the Multi2Sim environment) which reports a latency of 17 cycles. We then repeated the simulations in Multi2Sim with the new hardware latency for AES to compute the speedup.

The latency required to encrypt/decrypt a cache line that varies from 16 to 128 bytes, typically ranges from 11 to 160 cycles [117]. Thus, it is evident that encryption/decryption contribute to a large fraction of the memory authentication time. While existing works have assumed the use of a dedicated hardware module in the memory controller to perform the encryption/decryption task, our work shows the performance advantages of tightly coupling a hardware cryptographic module to the processor in the memory controller that can be invoked using custom instructions. In particular, we use of separate custom instructions for various encryption/decryption steps that are commonly used in cryptographic algorithms, i.e byte swapping, shift row. As such, the advantages of the proposed method is the flexibility of implementing different encryption/decryption techniques without the need for fabricating a new memory controller [118].

6.3.2 Performance Evaluation

We used Synopsys Design Compiler [119] to synthesize the AES RTL hardware description targeting a standard cell library (LSI-10k). In the custom instruction implementation, 32-bit inputs are passed sequentially to form 128 bit AES inputs. The AES encryption

consists of the following major functions: 1) Byte Substitution, 2) Shift Row, 3) Mix Column, 4) Add Round Key, 5) Key Expansion, and 6) Inverse Key Expansion. The proposed AES was designed as a 5-stage pipeline with a minimum clock period of 3ns. In the Multi2Sim framework with a 1GHz clock frequency, we fixed the encryption/decryption to 17 clock cycles. We ran the twelve benchmarks for two different tree architectures: 1) balanced TEC-tree implementation (BTEC) [33], 2) BTEC with AES as a CI, 3) proposed dynamic skewed tree [2], and 4) proposed dynamic skewed tree with AES as a CI. The authentication time for the benchmarks are shown in Fig. 6.16. The lightly shaded regions depict the authentication time with AES implemented as CI compared with a system where AES was implemented purely in software. We can observe that with AES implemented as a custom instruction, the proposed dynamic skewed memory integrity tree still exhibits better performance for the integrity check time compared to BTEC. The proposed method has, on average, 40% lower integrity check time over BTEC. There is an average improvement of 50% for the integrity check time when implementing AES as a custom instruction in almost all applications.

6.4 Security Analysis

In this section, we present the security analysis of the methods proposed so far including the static skewed memory integrity tree in Chapter 3, the dynamic skewed memory integrity tree in Chapter 4, the cache-aware dynamic skewed memory integrity tree in Chapter 5, and the methods introduced in this chapter. All these methods extend the TEC-tree which provides countermeasures for spoofing, splicing, and replay attacks.

Confidentiality: Similar to the TEC-tree, the proposed memory integrity tree relies on the AREA-Added Redundancy Explicit Authentication (AREA) [120] scheme. AREA is based on Shannon’s diffusion property [121] where redundancy is appended to the plaintext

(a)

(b)

(c)

(d)

(e)

(f)

(g)

(h)

Figure 6.16: Authentication time (in ms) for twelve different applications with varying cache configurations

message before encryption and a verification step is performed after decryption. In order to simultaneously authenticate and encrypt a message, redundant bits are appended at the end of the plaintext message before encryption. This way, the resultant encrypted block will depend on all the bits in the plaintext block including the redundant bits. The corruption of even a single bit in the ciphertext block will impact the decryption. The recipient can detect a malicious modification by comparing the result of the decrypted redundant bits with those in the plaintext. AREA schemes are efficient as they only perform a single pass over the data on encryption and decryption to provide for both data confidentiality and authenticity. In our scheme, we propose to leverage on Shannon's diffusion property by adding a nonce to the data to form a data chunk. Theoretically a block cipher must be indistinguishable (from an adversary standpoint) from a random permutation with equiprobable outputs i.e. the redundancy in the statistics of the plaintext block p has to be dissipated in the statistics of the ciphertext block c . Once a block encryption is performed, the resulting position and value of each bit in c is a function of all bits of the corresponding plaintext block. Thus, once the block is encrypted, the data and nonce cannot be differentiated. For an a -bit nonce, the probability that the last a bits remain the same after tampering is $1/(2^a)$.

We encrypt the data using the AES algorithm. We use a block size and key of 128 bits (probability of a successful attack is extremely low i.e., $1/2^{128}$). The key used for encryption is securely stored on-chip. The AES mode used is Electronic Code Block (ECB). This enables each block to be processed independently, thus reducing the granularity of integrity verification. Only one cipher block is loaded and decrypted for one load/store instruction. The only drawback is that it produces the same ciphered text each time for a particular data, but we overcome this limitation by using a nonce which makes each ciphered chunk unique.

Active Attacks: In our tree, a nonce is formed by concatenating address and count i.e. a -bit nonce = d -bits of address + r -bit of count. This ensures that the nonce is unique for each location. The probabilities of a successful bus attack for our given threat model are as shown in Table 6.3.

Table 6.3: Security limitations

Attacks	Spoofing	Splicing	Replay
Time (Sec)	$1/2^a$	0	$1/2^r$

Spoofing attacks are detected by making use of the block AREA scheme. The nonce is checked during the verification step after decryption. Any change in the data will be reflected, and the last a bits obtained would have changed. This mismatch would raise the alarm in the system, and the data will not be passed to the processor. The probability is derived directly from the use of block AREA scheme as explained earlier.

Splicing attacks are detected during the first stage of verification. As the address bits are stored in the nonce, if there is a mismatch between the address used to fetch the chunk and the bits extracted from the chunk, then the data would not be processed further, and an alarm would be raised. Thus, a 32-bit address space is completely protected from attacks if we allocate 32 bits to the address segment of the nonce.

Replay attacks are prevented due to the property of uniqueness of the nonce. If an address is replayed, the count values of the replayed and the current version will not match. The probability of a successful attack is directly dependent on the length of the count used in the nonce. The attack would be detected at the first non-replayed data block. If the entire tree is replayed, an alarm would be raised at the last verification step of matching the root node with the on-chip counter. DISSECT gives users the option of improving timing performance at the cost of reduced security. For all other proposed techniques, the replay security is as effective as a standard TEC-tree.

Preventing Counter Reuse: The security of encryption and integrity checks is retained with skewed trees, as we only change the tree representation without any modification to the encryption algorithm. By ensuring that the effective counter value is always forward-moving and never repeated, counter reuse is prevented.

Side-Channel Leakage: In side-channel attacks, the attacker measures side-channel information leakage such as power or timing to decipher the encryption keys being used for authentication. Recent works have demonstrated the feasibility of cache side-channel attacks on systems employing a standard memory integrity tree [122]. Cache-based side-channel analysis falls into one of two main categories: 1) trace-driven attacks, where individual cache hits and misses yield information leakage, and 2) time-driven attacks, where the aggregated number of cache misses and the execution time of the algorithm can be correlated to provide useful information. We wish to highlight that the security vulnerabilities of cache attacks on systems employing standard memory integrity tree used in prior works, and the proposed dynamic skewed integrity tree in this thesis is the same due to the following reasons. Firstly, the cache characteristics (hits and misses) will not differ in both methods as long as the grouping of data elements in leaf nodes are the same. This is because only the data elements are cached, and the contents of the integrity tree are stored in the external main memory. As such, compared to existing static integrity trees, the proposed skewed integrity tree only reduces the number of external memory accesses taken to traverse the integrity tree for authentication in the event of a cache miss. Secondly, the success of a side-channel attack is independent of the structural format of the tree as the tree does not play a role in encrypting/decrypting the keys. The encryption/decryption of keys is carried out by the cryptographic algorithm in the memory controller. We have used the standard AES algorithm, where the execution and processing times are not affected by the dynamic restructuring of the tree.

Key Management: We assume a trusted operating system and a scheme that can securely manage keys and ensure they are not compromised. Our contributions are orthogonal to the choice of the key management scheme and the trusted platform used, and can complement existing countermeasures against software attacks.

Resilience to Denial of Service: Overflowing of counters can flood the memory system with accesses for re-encryption resulting in large delays. In DISSECT, we avoid counter overflow with the proposed splitting mechanism, thus shielding the processor from overflow-handling accesses.

6.5 Summary

In this chapter, we proposed three techniques to improve scalability of dynamic skewed memory integrity trees. For CADST framework, the integrity tree is initialized with a vEB organization to reduce the memory-cache accesses and the skewing algorithm shifts memory blocks which form sub-trees so as to maintain minimal number of cache misses during tree traversal. We then presented DISSECT, a skewed integrity tree which can be customized to increase arity depending on the security and performance requirements. We propose a technique to address local counter overflows by dynamically splitting only those nodes whose local counter is about to exceed its maximum permissible value. Lastly, we evaluate the performance benefits of realizing AES encryption using custom instructions, where a hardware accelerator is tightly coupled with a memory controller to improve runtime verification overheads. It is noteworthy that these techniques are complementary to each other and can be deployed accordingly based on the available hardware support on a target platform. The security of the proposed methods against splicing, spoofing and replay attacks is discussed.

The evaluations and results for the research undertaken in this thesis clearly demonstrate that the proposed techniques can improve performance of memory authentication

over the prevalent approaches that rely on a balanced memory integrity tree. In the next chapter, we will conclude the thesis and discuss a several directions for future work.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this thesis, we proposed architecture-aware algorithms to support the adaptation of memory integrity trees to application workloads in order to reduce the authentication time. The proposed methods overcome the performance overhead limitations of existing schemes by enabling the more frequently accessed memory blocks in the memory integrity tree to be shifted closer to the root so that they traverse shorter verification paths. Two different tree structures have been proposed i.e., static skewed tree and dynamic skewed tree. In the former, the memory integrity tree is customized at compile-time based on static analysis of the memory access patterns, while the latter enables the tree to be dynamically skewed based on the runtime memory access patterns. We also presented techniques to optimize the performance the proposed dynamic memory integrity tree approach by exploiting the cache properties of the system and incorporating further tree adjustment to increase the tree arity. We also evaluated the benefits of incorporating custom hardware cryptographic modules in the memory controller for accelerating memory authentication. Experimental results based on widely used benchmarks demonstrate the effectiveness of the proposed methods. We also performed security analysis to show that the performance gain of the proposed methods can be achieved without compromising on the security strength.

In Chapter 3, we presented a framework for generating a customized skewed memory integrity tree based on the frequency of memory accesses of the application. We modified the TEC-tree to allow for the more frequently accessed memory blocks to traverse shorter verification paths. This is achieved by including additional fields in the data chunk to indicate the parent of the leaf nodes. We formulated the weights to reflect the memory access overheads of read and write operations of each memory block. The package-merge algorithm was adapted to place the weighted memory blocks in the skewed tree so that memory blocks with larger weights will require fewer verification steps.

In Chapter 4, we propose an algorithm to construct a dynamic skewed memory integrity tree. The tree is dynamically restructured based on runtime memory access patterns. Data symbols that have the same frequency are placed together in a single node on the tree. Nodes are relocated at runtime to make sure that the nodes with higher frequency are positioned closer to the root. This reduces the time required for runtime verification of the data accessed from the external memory. Experimental results show improvement in execution time when compared to the conventional balanced tree approach.

Next, in Chapter 5, we propose a framework for dynamic skewed memory integrity tree that takes advantage of the cache properties to improve performance. The framework combines architecture-specific optimization of the integrity tree structure with runtime mechanisms to minimize the authentication overhead. Compared to existing integrity tree methods that are also based on exploiting memory access patterns, the proposed integrity tree can adapt more effectively to the dynamic memory access patterns by maintaining a separate *counter* for each leaf node. We also show that the proposed framework can minimize energy consumption for a given cache configuration. Extensive simulations on systems with varying cache configurations using applications from SPEC-CPU2006, SPLASH-2, and PARSEC benchmarks show that the proposed framework leads to a significant reduction in authentication time compared to existing methods.

Finally in Chapter 6, we presented three different schemes to increase the scalability of the proposed dynamic integrity tree for applications with large number of memory accesses. We explored efficient cache-aware algorithms for caching the nodes of the dynamic skewed tree on a dedicated on-chip Tree-Cache. The tree layout is initialized with a vEB organization to reduce the cache misses. We also presented DISSECT, a dynamic skewed integrity tree structure that enables the tree arity to be increased at runtime to further reduce the authentication overhead. This can be achieved with a novel technique to prevent local counter overflows when the tree is split. Finally, we quantify the benefits of implementing AES as a custom instruction for executing the cryptographic operations during memory authentication.

7.2 Future Work

In this thesis, we have ascertained the performance benefits of employing dynamic skewed integrity trees for memory authentication. In addition, in Chapter 6, we have performed security analysis to show that the proposed methods do not lead to increase in vulnerability to cache attacks that are aimed at inferring the secret cryptographic key. Moreover, in our work published in [5], we have demonstrated how the dynamic skewed tree can be used for detection of RowHammer attacks. In this section, we will discuss four future research directions of this thesis. The first aims to implement a programmable memory controller that realizes the proposed dynamic skewed memory integrity framework on FPGA. The second and third future directions aim to extend the proposed methods for multi-core systems in hosted environments (with Operating Systems). Finally, the fourth research direction aims to study the security vulnerabilities of the proposed methods in reverse engineering attacks.

7.2.1 Energy-Efficient Secure Memory Controller

As part of our future work, we will develop a programmable memory controller that realizes the proposed dynamic skewed memory integrity tree framework. We plan to integrate the memory controller with the RISC-V soft-processor [123] on a FPGA system to demonstrate the energy benefits of the proposed integrity tree that adapts to the memory workloads. As discussed in the previous section, the utilization of custom instruction for AES encryption/decryption reduces the authentication time substantially. As such, we will also augment the memory controller with a tightly-coupled hardware AES module to execute the encryption/decryption steps during memory authentication. In addition, we plan to explore the benefits of implementing the tree traversal and restructuring algorithms in hardware.

7.2.2 Memory Authentication in Hosted Environments

Currently, all the evaluations in our thesis have been undertaken on a stand-alone (bare-metal) environment without an operating system (OS). While many embedded systems operate in a bare-metal environment, more complex embedded applications that involve multi-tasking will require an Real-Time Operating System (RTOS). We plan to extend the proposed dynamic memory integrity tree methods to embedded systems with RTOS. This poses a challenge for incorporating a memory integrity tree that can adapt to the workloads of the various concurrent tasks, when the RTOS switches between multiple tasks based on incoming events and priorities.

7.2.3 Dynamic Integrity Tree for Multi-Core Platforms

In this thesis we have evaluated our work on a single-core system. With the increasing number of processor cores in System-on-a-Chip, there is also a need to develop lightweight memory authentication schemes for multi-core systems. This line of research has been

largely neglected. The work undertaken by [124] reported that the bonsai Merkle tree exhibited performance degradation on a multi-core system when compared to a single-core system due to memory bandwidth limitations. In our future work, we plan to investigate the feasibility of implementing a dynamic integrity tree in the shared memory that can adapt to workloads running on the various cores in order to reduce the authentication overhead. We expect a number of new challenges that has to be addressed on the multi-core platform. For example, a multi-core system with shared memory would need to maintain memory coherency and this must be taken into account while the integrity tree adapts to the workload on different cores. Similar to the objectives of this thesis, the proposed schemes must also lead to overall improvement in performance and energy efficiency.

7.2.4 Mitigating Reverse Engineering Attacks

Our proposed memory authentication framework enables the integrity tree to be dynamically skewed which resulted in an overall reduction in instruction per cycle and energy overhead compared with the existing methods. We have also established in this thesis that the proposed dynamic integrity tree have the same security strength as conventional integrity trees with respect to cache side-channel attacks that aim to recover the secret cryptographic key. However, the dynamically skewing operations of the integrity tree inadvertently reveal information about the implementation of the algorithm, which may be an Intellectual Property (IP) that need to be kept undisclosed. In our future work, we will investigate efficient methods to mask the information disclosure of the algorithm running on a system that employs the dynamically skewed integrity tree in order to protect against reverse engineering attacks. To achieve this, we can leverage on the ability of our scheme to dynamically restructure the tree to produce random memory access patterns. The challenge is to ensure strong memory protection while keeping the overall memory authentication overheads low.

References

- [1] S. Vig, T. Y. Tzer, G. Jiang, and S.-K. Lam, “Customizing skewed trees for fast memory integrity verification in embedded systems,” in *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 2017, pp. 213–218.
- [2] S. Vig, G. Jiang, and S.-K. Lam, “Dynamic skewed tree for fast memory integrity verification,” in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 642–647.
- [3] S. Vig, R. Juneja, G. Jiang, S.-K. Lam, and C. Ou, “Framework for fast memory authentication using dynamically skewed integrity tree,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2019.
- [4] S. Vig, R. Juneja, and S.-K. Lam, “Dissect: Dynamic skew-and-split tree for memory authentication,” in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*.
- [5] S. Vig, S. Bhattacharya, D. Mukhopadhyay, and S.-K. Lam, “Rapid detection of rowhammer attacks using dynamic skewed hash tree,” in *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, 2018, p. 7.
- [6] P. B. Ankita Bhutani, “Embedded System Market Size By Application,” <https://www.gminsights.com/industry-analysis/embedded-system-market>, 2016, [Online].
- [7] R. Elbaz, D. Champagne, C. Gebotys, R. B. Lee, N. Potlapally, and L. Torres, “Hardware mechanisms for memory authentication: A survey of existing techniques and engines,” in *Transactions on Computational Science IV*. Springer, 2009.

REFERENCES

- [8] F. I. A. Shaikh, “Security framework for the internet of things leveraging network telescopes and machine learning,” Ph.D. dissertation, University of South Florida, 2019.
- [9] J. Portillo, E. John, and S. Narasimhan, “Building trust in 3pip using asset-based security property verification,” in *2016 IEEE 34th VLSI Test Symposium (VTS)*. IEEE, 2016, pp. 1–6.
- [10] M. Keefe, “Timeline: Critical infrastructure attacks increase steadily in past decade,” *Computerworld*, 2012.
- [11] G. Alterson, “Confronting One Of Healthcare’s Biggest Challenges: Cyber Risk,” <https://www.forbes.com/sites/insights-intelai/2019/02/11/confronting-one-of-healthcares-biggest-challenges-cyber-risk/#125518e17b83>, 2019, [Online].
- [12] S. D’Alfonso, “4.5 Million Patient Records Stolen in CHS Data Breach,” <https://securityintelligence.com/4-5-million-patient-records-stolen-in-chs-data-breach-whats-next/>, 2014, [Online].
- [13] A. Nahapetian, “Side-channel attacks on mobile and wearable systems,” in *2016 13th IEEE Annual Consumer Communications & Networking Conference (CCNC)*. IEEE, 2016, pp. 243–247.
- [14] D. Halperin, T. S. Heydt-Benjamin, B. Ransford, S. S. Clark, B. Defend, W. Morgan, K. Fu, T. Kohno, and W. H. Maisel, “Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses,” in *2008 IEEE Symposium on Security and Privacy (sp 2008)*. IEEE, 2008, pp. 129–142.
- [15] C. Li, A. Raghunathan, and N. K. Jha, “Hijacking an insulin pump: Security attacks and defenses for a diabetes therapy system,” in *2011 IEEE 13th International Conference on e-Health Networking, Applications and Services*. IEEE, 2011, pp. 150–156.

REFERENCES

- [16] J. Liu and W. Sun, “Smart attacks against intelligent wearables in people-centric internet of things,” *IEEE Communications Magazine*, vol. 54, no. 12, pp. 44–49, 2016.
- [17] R. Westervelt, “The Total Global Cost Of Cybercrime \$400 Billion A Year And Growing,” <https://www.crn.com/news/security/300073063/the-total-global-cost-of-cybercrime-400-billion-a-year-and-growing.htm>, 2014, [Online].
- [18] T. Bradley, “Your business cant afford the cost of cyber crime,” <https://www.csoononline.com/article/2837805/your-business-can-t-afford-the-cost-of-cyber-crime.html>, 2014, [Online].
- [19] D. N. Serpanos and A. G. Voyiatzis, “Security challenges in embedded systems,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 12, no. 1s, p. 66, 2013.
- [20] R. Anderson, *Security engineering*. John Wiley & Sons, 2008.
- [21] S. Thom, J. Cox, D. Linsley, M. Nystrom, H. Raj, D. Robinson, S. Saroiu, R. Spiger, and A. Wolman, “Firmware-based trusted platform module for arm processor architectures and trustzone security extensions,” Feb. 12 2013, uS Patent 8,375,221.
- [22] V. Jyothi, M. Thoonoli, R. Stern, and R. Karri, “Fpga trust zone: Incorporating trust and reliability into fpga designs,” in *2016 IEEE 34th International Conference on Computer Design (ICCD)*. IEEE, 2016, pp. 600–605.
- [23] A. P. Fournaris and N. Sklavos, “Secure embedded system hardware design—a flexible security and trust enhanced approach,” *Computers & Electrical Engineering*, vol. 40, no. 1, pp. 121–133, 2014.
- [24] Arm trust zone. [Online]. Available: <http://www.arm.com/products/processors/technologies/trustzone/index.ph>
- [25] O. Goldreich and R. Ostrovsky, “Software protection and simulation on oblivious rams,” *Journal of the ACM (JACM)*, vol. 43, no. 3, pp. 431–473, 1996.

REFERENCES

- [26] R. Strackx, F. Piessens, and B. Preneel, “Efficient isolation of trusted subsystems in embedded systems,” in *International Conference on Security and Privacy in Communication Systems*. Springer, 2010, pp. 344–361.
- [27] G. Liu, X. Zhang, D. Wang, Z. Liu, and H. Wang, “Security memory system for mobile device or computer against memory attacks,” in *International Conference on Trustworthy Computing and Services*. Springer, 2013, pp. 1–8.
- [28] D. Papp *et al.*, “Embedded systems security: Threats, vulnerabilities, and attack taxonomy,” in *Privacy, Security and Trust (PST), 2015 13th Annual Conference on*. IEEE, 2015.
- [29] P. Koopman, “Embedded system security,” *Computer*, vol. 37, no. 7, pp. 95–97, 2004.
- [30] N. R. Potlapally, S. Ravi, A. Raghunathan, and N. K. Jha, “Analyzing the energy consumption of security protocols,” in *Proceedings of the 2003 international symposium on Low power electronics and design*. ACM, 2003, pp. 30–35.
- [31] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, “Architectural support for copy and tamper resistant software,” *ACM SIGPLAN Notices*, vol. 35, no. 11, pp. 168–177, 2000.
- [32] S. A. Rotondo, “Trusted computing group,” in *Encyclopedia of Cryptography and Security*. Springer, 2011, pp. 1331–1331.
- [33] R. Elbaz *et al.*, “Tec-tree: A low-cost, parallelizable tree for efficient defense against memory replay attacks,” in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2007.
- [34] S. Ravi, A. Raghunathan, and S. Chakradhar, “Tamper resistance mechanisms for secure embedded systems,” in *VLSI Design, 2004. Proceedings. 17th International Conference on*. IEEE, 2004, pp. 605–611.
- [35] P. KUDUMAKIS, “Moses: Mpeg open security for embedded systems,” in *Proceedings of the 4th International Workshop on Image Analysis for Multimedia Interactive Services (WIAMIS’03), London, UK, 2003*.

REFERENCES

- [36] S. Gueron, “A Memory Encryption Engine Suitable for General Purpose Processors,” 2016.
- [37] S. Chhabra and Y. Solihin, “Green secure processors: towards power-efficient secure processor design,” in *Transactions on computational science X*. Springer, 2010.
- [38] W. J. Dally, J. Balfour, D. Black-Shaffer, J. Chen, R. C. Harting, V. Parikh, J. Park, and D. Sheffield, “Efficient embedded computing,” *Computer*, no. 7, pp. 27–32, 2008.
- [39] K. N. McGill, “Trusted mobile devices: Requirements for a mobile trusted platform module,” *Johns hopkins apl technical digest*, vol. 32, no. 2, p. 544, 2013.
- [40] M. E. Donaldson, “Inside the buffer overflow attack: Mechanism, method & prevention,” *GSEC Version*, vol. 1, no. 3, p. 5, 2002.
- [41] “Apple ipods with virus,” <https://techcrunch.com/2006/10/17/apple-ships-small-number-of-55g-ipods-with-windows-virus-wtf/>.
- [42] T. Abera, N. Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadeghi, and G. Tsudik, “C-flat: control-flow attestation for embedded systems software,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 743–754.
- [43] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, “Data-oriented programming: On the expressiveness of non-control data attacks,” in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 969–986.
- [44] D. Brumley and D. Boneh, “Remote timing attacks are practical,” *Computer Networks*, vol. 48, no. 5, pp. 701–716, 2005.
- [45] T. S. Messerges, E. A. Dabbish, and R. H. Sloan, “Examining smart-card security under the threat of power analysis attacks,” *IEEE transactions on computers*, vol. 51, no. 5, pp. 541–552, 2002.
- [46] Y. Hori, T. Katashita, A. Sasaki, and A. Satoh, “Electromagnetic side-channel attack against 28-nm fpga device,” *Pre-proceedings of WISA*, p. 84, 2012.

REFERENCES

- [47] R. Carbone, C. Bean, and M. Salois, “An in-depth analysis of the cold boot attack: Can it be used for sound forensic memory acquisition?” DEFENCE RESEARCH AND DEVELOPMENT CANADA VALCARTIER (QUEBEC), Tech. Rep., 2011.
- [48] D. Petryk, Z. Dyka, and P. Langendoerfer, “Optical fault injections: a setup comparison.”
- [49] “Why every chip can be hacked with this tool,” <https://semiengineering.com/every-chip-can-be-hacked-with-this-tool/>.
- [50] S. Govindavajhala and A. W. Appel, “Using memory errors to attack a virtual machine,” in *2003 Symposium on Security and Privacy, 2003*. IEEE, 2003, pp. 154–165.
- [51] P. Mishra, S. Bhunia, and M. Tehranipoor, *Hardware IP security and trust*. Springer, 2017.
- [52] G. Canivet, R. Leveugle, J. Clediere, F. Valette, and M. Renaudin, “Characterization of effective laser spots during attacks in the configuration of a virtex-ii fpga,” in *2009 27th IEEE VLSI Test Symposium*. IEEE, 2009, pp. 327–332.
- [53] P. Kocher, R. Lee, G. McGraw, A. Raghunathan, and S. Moderator-Ravi, “Security as a new dimension in embedded system design,” in *Proceedings of the 41st annual Design Automation Conference*. ACM, 2004, pp. 753–760.
- [54] S. Tunney and J. Stille, “Palladium-catalyzed coupling of aryl halides with (trimethylstannyl) diphenylphosphine and (trimethylsilyl) diphenylphosphine,” *The Journal of Organic Chemistry*, vol. 52, no. 5, pp. 748–753, 1987.
- [55] M. Peinado, Y. Chen, P. England, and J. Manferdelli, “Ngsch: A trusted open system,” in *Australasian Conference on Information Security and Privacy*. Springer, 2004, pp. 86–97.
- [56] R. Affeldt and N. Marti, “Formal verification of arithmetic functions in smartmips assembly,” in *23rd Workshop of the Japan Society for Software Science and Technology (JSSST 2006)*.

REFERENCES

- [57] G. E. Suh *et al.*, “Aegis: A single-chip secure processor,” *Information Security Technical Report*, vol. 10, 2005.
- [58] M. Chew and D. Song, “Mitigating buffer overflows by operating system randomization,” 2002.
- [59] M. Frantzen and M. Shuey, “Stackghost: Hardware facilitated stack protection.” in *USENIX Security Symposium*, vol. 112, 2001.
- [60] S. Designer, “Non-executable stack patch,” 1997.
- [61] E. Hess, N. Janssen, B. Meyer, and T. Schütze, “Information leakage attacks against smart card implementations of cryptographic algorithms and countermeasures—a survey,” in *EUROSMART Security Conference*, vol. 130. Citeseer, 2000.
- [62] D. Lie, C. A. Thekkath, and M. Horowitz, “Implementing an untrusted operating system on trusted hardware,” in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003, pp. 178–192.
- [63] Y. Chen, R. Venkatesan, M. Cary, R. Pang, S. Sinha, and M. H. Jakubowski, “Oblivious hashing: A stealthy software integrity verification primitive,” in *International Workshop on Information Hiding*. Springer, 2002, pp. 400–414.
- [64] E. F. Brickell, C. D. Hall, J. F. Cihula, and R. Uhlig, “Method of improving computer security through sandboxing,” Mar. 15 2011, uS Patent 7,908,653.
- [65] S. A. Hofmeyr, S. Forrest, and A. Somayaji, “Intrusion detection using sequences of system calls,” *Journal of computer security*, vol. 6, no. 3, pp. 151–180, 1998.
- [66] O. Kömmerling and M. G. Kuhn, “Design principles for tamper-resistant smartcard processors.” *Smartcard*, vol. 99, pp. 9–20, 1999.
- [67] E. Sprunk, “Clock frequency modulation for secure microprocessors,” Apr. 4 1995, uS Patent 5,404,402.
- [68] K. Tiri, “Side-channel attack pitfalls,” in *2007 44th ACM/IEEE Design Automation Conference*. IEEE, 2007, pp. 15–20.

REFERENCES

- [69] S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi, “Towards sound approaches to counteract power-analysis attacks,” in *Annual International Cryptology Conference*. Springer, 1999, pp. 398–412.
- [70] D. May, H. L. Muller, and N. P. Smart, “Random register renaming to foil dpa,” in *CHES*, vol. 1. Springer, 2001, pp. 28–38.
- [71] J.-J. Quisquater and D. Samyde, “Electromagnetic analysis (ema): Measures and counter-measures for smart cards,” *Smart Card Programming and Security*, pp. 200–210, 2001.
- [72] R. Elbaz, L. Torres, G. Sassatelli, P. Guillemin, M. Bardouillet, and A. Martinez, “A parallelized way to provide data encryption and integrity checking on a processor-memory bus,” in *Proceedings of the 43rd annual Design Automation Conference*. ACM, 2006.
- [73] S. Nimgaonkar and M. Gomathisankaran, “Energy efficient memory authentication mechanism in embedded systems,” in *Electronic System Design (ISED), 2011 International Symposium on*. IEEE, 2011, pp. 248–253.
- [74] A. M. Fiskiran and R. B. Lee, “Runtime execution monitoring (rem) to detect and prevent malicious code execution,” in *Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings. IEEE International Conference on*. IEEE, 2004, pp. 452–457.
- [75] T. S. Lehman *et al.*, “PoisonIvy: Safe speculation for secure memory,” in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, oct 2016.
- [76] S. H. Kim *et al.*, “Fully Batch Processing Enabled Memory Integrity Verification Algorithm Based on Merkle Tree.” Springer, Cham, 2016.
- [77] Microsoft palladium: Next generation secure computing base. [Online]. Available: <https://epic.org/privacy/consumer/microsoft/palladium.html>

- [78] T. Müller and M. Spreitzenbarth, “Frost,” in *International Conference on Applied Cryptography and Network Security*. Springer, 2013, pp. 373–388.
- [79] S. Ravi, S. Ravi, A. Raghunathan, P. Kocher, and S. Hattangady, “Security in embedded systems: Design challenges,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 3, no. 3, pp. 461–491, 2004.
- [80] J. Yang, Y. Zhang, and L. Gao, “Fast secure processor for inhibiting software piracy and tampering,” in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2003, p. 351.
- [81] P. Colp, J. Zhang, J. Gleeson, S. Suneja, E. de Lara, H. Raj, S. Saroiu, and A. Wolman, “Protecting data on smartphones and tablets from memory attacks,” *ACM SIGPLAN Notices*, vol. 50, no. 4, pp. 177–189, 2015.
- [82] N. Heninger, “Cold-boot attacks,” in *Encyclopedia of Cryptography and Security*. Springer, 2011, pp. 216–217.
- [83] O. Cherednichenko, A. Baranov, and T. Morozova, “Side-channel attack,” 2013.
- [84] D. Gruss, C. Maurice, and S. Mangard, “Rowhammer. js: A remote software-induced fault attack in javascript,” in *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2016, pp. 300–321.
- [85] M. Becher, M. Dornseif, and C. N. Klein, “Firewire: all your memory are belong to us,” *Proceedings of CanSecWest*, p. 67, 2005.
- [86] A. Francillon and C. Castelluccia, “Code injection attacks on harvard-architecture devices,” in *Proceedings of the 15th ACM conference on Computer and communications security*. ACM, 2008, pp. 15–26.
- [87] J. Crenne, R. Vaslin, G. Gogniat, J.-P. Diguët, R. Tessier, and D. Unnikrishnan, “Configurable memory security in embedded systems,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 12, no. 3, p. 71, 2013.

REFERENCES

- [88] M.-K. Yoon, S. Mohan, J. Choi, and L. Sha, “Memory heat map: anomaly detection in real-time embedded systems using memory behavior,” in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2015, pp. 1–6.
- [89] P. Henderson and M. Maheswaran, “Chaotic memory randomization for securing embedded systems,” *arXiv preprint arXiv:1611.00742*, 2016.
- [90] F. Liu, H. Wu, K. Mai, and R. B. Lee, “Newcache: Secure cache architecture thwarting cache side-channel attacks,” *IEEE Micro*, vol. 36, no. 5, pp. 8–16, 2016.
- [91] R. Vaslin, G. Gogniat, J.-P. Diguët, E. Wanderley, R. Tessier, and W. Burleson, “A security approach for off-chip memory in embedded microprocessor systems,” *Microprocessors and Microsystems*, vol. 33, no. 1, pp. 37–45, 2009.
- [92] T. Liu, H. Guo, S. Parameswaran, and X. S. Hu, “Improving tag generation for memory data authentication in embedded processor systems,” in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2016, pp. 50–55.
- [93] X. Zhuang, T. Zhang, and S. Pande, “Hide: an infrastructure for efficiently protecting information leakage on the address bus,” in *ACM SIGPLAN Notices*, vol. 39, no. 11. ACM, 2004, pp. 72–84.
- [94] G. Duc and R. Keryell, “Cryptopage: an efficient secure architecture with memory encryption, integrity and information leakage protection,” in *2006 22nd Annual Computer Security Applications Conference (ACSAC’06)*. IEEE, 2006, pp. 483–492.
- [95] R. C. Merkle, “Protocols for public key cryptosystems.” in *IEEE Symposium on Security and privacy*, vol. 122, 1980.
- [96] W. E. Hall and C. S. Jutla, “Parallelizable authentication trees,” in *International Workshop on Selected Areas in Cryptography*. Springer, 2005, pp. 95–109.
- [97] G. Saileshwar, P. Nair, P. Ramrakhiani, W. Elsasser, J. Joao, and M. Qureshi, “Morphable counters: Enabling compact integrity trees for low-overhead secure memories,” in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 416–427.

REFERENCES

- [98] M. Taassori, A. Shafiee, and R. Balasubramonian, “Vault: Reducing paging overheads in sgx with efficient integrity verification structures,” in *ACM SIGPLAN Notices*, vol. 53, no. 2. ACM, 2018, pp. 665–678.
- [99] J. Rakshit and K. Mohanram, “Assure: Authentication scheme for secure energy efficient non-volatile memories,” in *Design Automation Conference (DAC), 2017 54th ACM/EDAC/IEEE*. IEEE, 2017, pp. 1–6.
- [100] D. C. Suresh, W. A. Najjar, and J. Yang, “Power efficient instruction caches for embedded systems,” in *International Workshop on Embedded Computer Systems*. Springer, 2005.
- [101] B. Rogers, S. Chhabra, M. Prvulovic, and Y. Solihin, “Using address independent seed encryption and bonsai merkle trees to make secure processors os-and performance-friendly,” in *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*. IEEE, 2007, pp. 183–196.
- [102] B. Gassend *et al.*, “Caches and hash trees for efficient memory integrity verification,” in *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on*. IEEE, 2003.
- [103] F. S. Solutions, “Inc.,” nios ii software user guide-system analyzer for the nios ii processor core,” 2004.
- [104] L. L. Larmore and D. S. Hirschberg, “A fast algorithm for optimal length-limited huffman codes,” *Journal of the ACM (JACM)*, vol. 37, no. 3, 1990.
- [105] S. Nagaraj, “Optimal binary search trees,” *Theoretical Computer Science*, vol. 188, 1997.
- [106] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii, “Chstone: A benchmark program suite for practical c-based high-level synthesis,” in *2008 IEEE International Symposium on Circuits and Systems*. IEEE, 2008, pp. 1192–1195.
- [107] “Snu real-time benchmarks,” <http://www.cprover.org/goto-cc/examples/snu.html>, accessed: 2010-09-30.

REFERENCES

- [108] S. Pigeon and Y. Bengio, “A memory-efficient adaptive huffman coding algorithm for very large sets of symbols,” in *Data Compression Conference, 1998. DCC’98. Proceedings.* IEEE, 1998, p. 568.
- [109] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, “Multi2sim: a simulation framework for cpu-gpu computing,” in *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT).* IEEE, 2012, pp. 335–344.
- [110] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, “Mcpat: an integrated power, area, and timing modeling framework for multi-core and manycore architectures,” in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture.* ACM, 2009, pp. 469–480.
- [111] V. Costan and S. Devadas, “Intel sgx explained.” *IACR Cryptology ePrint Archive*, vol. 2016, no. 086, pp. 1–118, 2016.
- [112] F. Huang, D. Feng, Y. Hua, and W. Zhou, “A wear-leveling-aware counter mode for data encryption in non-volatile memories,” in *Proceedings of the Conference on Design, Automation & Test in Europe.* European Design and Automation Association, 2017, pp. 910–913.
- [113] C. Yan, D. Engler, M. Prvulovic, B. Rogers, and Y. Solihin, “Improving cost, performance, and security of memory encryption and authentication,” in *ACM SIGARCH Computer Architecture News*, vol. 34, no. 2. IEEE Computer Society, 2006, pp. 179–190.
- [114] M. N. Bojnordi and E. Ipek, “Pardis: A programmable memory controller for the ddrx interfacing standards,” in *2012 39th Annual International Symposium on Computer Architecture (ISCA).* IEEE, 2012, pp. 13–24.
- [115] J. Nechvatal, E. Barker, L. Bassham, W. Burr, M. Dworkin, J. Foti, and E. Roback, “Report on the Development of the Advanced Encryption Standard (AES),” 2000.

REFERENCES

- [116] H. Li and Z. Friggstad, “An efficient architecture for the aes mix columns operation,” in *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on*. IEEE, 2005.
- [117] M. Henson and S. Taylor, “Memory encryption: A survey of existing techniques,” *ACM Computing Surveys (CSUR)*, vol. 46, no. 4, p. 53, 2014.
- [118] R. R. Taylor and S. C. Goldstein, “A high-performance flexible architecture for cryptography,” in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 1999, pp. 231–245.
- [119] S. D. Compiler, “User manual for synopsys toolset version 2004.06,” *Synopsys, Inc*, 2004.
- [120] C. Fruhwirth, *New methods in hard disk encryption*. na, 2005.
- [121] C. E. Shannon, “A mathematical theory of cryptography,” *Memorandum MM*, vol. 45, 1945.
- [122] T. Unterluggauer, M. Werner, and S. Mangard, “Meas: memory encryption and authentication secure against side-channel attacks,” *Journal of cryptographic engineering*, vol. 9, no. 2, pp. 137–158, 2019.
- [123] Y. Lee, A. Waterman, R. Avizienis, H. Cook, C. Sun, V. Stojanović, and K. Asanović, “A 45nm 1.3 ghz 16.7 double-precision gflops/w risc-v processor with vector accelerators,” in *ESSCIRC 2014-40th European Solid State Circuits Conference (ESSCIRC)*. IEEE, 2014, pp. 199–202.
- [124] J. Lee, T. Kim, and J. Huh, “Reducing the memory bandwidth overheads of hardware security support for multi-core processors,” *IEEE Transactions on Computers*, vol. 65, no. 11, pp. 3384–3397, 2016.